程序设计语言 PASCAL

N. 沃思　K. 詹森 著

俞嘉惠 译

科 学 出 版 社

1983

内 容 简 介

PASCAL 语言是基于程序设计方法论的原则设计的一种语言，语法简单、数据结构和程序结构比较丰富，可用于教学、系统程序设计和科学计算。

本书译自 "PASCAL User Manual and Report" 一书和 "The Design of a PASCAL Compiler" 一文，全面介绍了 PASCAL 语言的报告、特点及其程序设计方法，并介绍了编译系统 PASCAL 6000-3.4 的特点、使用方法和实现方法。

本书可供计算机软件工作者和有关专业人员参考。

Kathleen Jensen, Niklaus Wirth

PASCAL USER MANUAL AND REPORT

Springer-Verlag, 1978

—AND—

Niklaus Wirth The Design of a PASCAL Compiler

程序设计语言 PASCAL

N. 沃思 K. 詹森 著

俞嘉惠 译

责任编辑 杨家福 刘晓融

科 学 出 版 社 出 版

北京朝阳门内大街137号

中国科学院印刷厂印刷

新华书店北京发行所发行 各地新华书店经售

*

1982年7月第一版 开本：787×1092 1/32

1982年7月第一次印刷 印张：7 7/8

印数：0001—18,950 字数：176,000

统一书号：15031·518

本社书号：3157·15—8

定价：1.25 元

译 者 的 话

七十年代以来，计算机软硬件管理的复杂性日益突出，在语言设计方面出现了以程序设计方法论为设计原则的倾向．这种倾向试图通过合理地选取、调整和安排语言内部的成分，采用模块化和可验证的约束规则等办法，从本质上改进软件复杂性的管理．

程序设计语言 PASCAL 就是基于这种思想设计的，它所关心的是概念、使用和实现等方面的简化．整个语言紧凑整齐，概念简洁，数据结构和控制结构比较丰富，目标程序效率较高，查错能力强．用 PASCAL 书写程序有助于提高程序的可读性、正确性和可靠性，有助于培养好的程序设计风格．它原来是作为教学用语言设计的，但在系统程序设计、程序验证和普通的程序设计等方面，都取得了一定的成功．由于这些特点，PASCAL 语言被公认为七十年代比较有影响的语言，目前正在为 PASCAL 制订国际标准．人们还以 PASCAL 为基础发展了一些重要语言，如并发 PASCAL、EUCLID、MODULA、MODULA2 等并发处理语言，特别是美国国防部正在研制的全军公共语言 ADA．

正如对其他语言的评价存在分歧那样，对 PASCAL 的评价也有分歧．其意见有的属于 PASCAL 本身的，如概念的抽象是否恰当，成分的取舍是否合理，语言定义尚欠严密等，有的则关系到程序设计语言的发展方向．

翻译本书的目的在于比较全面地介绍 PASCAL 语言，为此，除选译了 "PASCAL User Manual and Report" 一书的全

部内容作为本书的第一、二部分外，还选译了 N. Wirth 的“The Design of a PASCAL Compiler”一文作为第三部分；本书就是由这三部分组成的. 第一部分是对 PASCAL 语言的完整定义，并说明了它的设计目标. 第二部分是 CDC 6000 机 PASCAL 语言的使用手册，详尽地讲述了该语言各种特色，如何用 PASCAL 编写程序以及 PASCAL 的编译程序的功能与使用方法. 第三部分介绍了这个编译程序的设计思想和实现方法. 全书贯穿了作者在程序设计语言、编译程序的设计、以及程序设计方法论等方面的看法，这在七十年代是很有代表性的.

为了前后一致，全书已统一编目. 为阅读方便，本书将语言报告提到前面作为第一部分. 凡原文有欠妥之处，译文均予以更正，重要之处还加了注释. 由于前两部分原文出版比第三部分晚，其间语言有所更动，故本书前后有些内容不完全一致；另外，因某些名词术语尚未统一，只好按原文酌情拟定.

译者非常感谢七四九部队的徐仁尧、中国科学院计算技术研究所唐稚松、董韫美、曹东启、王廷俊、曹右琦以及上海计算技术研究所朱三元等同志，他们在本稿的译校过程中曾给予了许多帮助，有的仔细地校阅了译稿，有的提供了宝贵的意见. 由于译者水平有限，错误和不当之处在所难免，敬请读者批评指正.

序　　言[1)]

程序设计语言 PASCAL 的初稿是在 1968 年拟订的，它是从 ALGOL-60 和 ALGOL-W 型语言发展而来的. 经过广泛具体的研制之后，第一个编译程序于 1970 年开始运行，并在一年后发表了这一结果(见参考文献[1]和[8]). 随着其他计算机对研制编译程序的兴趣日益增加,要求加强PASCAL语言的研究;而且,两年来使用该语言的经验也表明有必要作些修改. 于是,在 1973 年发表了一个修订报告和一个用 ISO 字符集表示的语言定义.

这本小册子由两个部分组成: 用户手册和修订报告. **手册**供已掌握了一定的程序设计技巧并希望熟悉 PASCAL 语言的人们使用. 因此,这个手册是按照教材形式编写的,它用大量例子来说明 PASCAL 语言的各种特色. 概括性的表格和语法规范作为附录附在后面. 修订**报告**之所以收进本书，其目的是要为程序员和实现人员提供一份简明的、基本的参考材料. 它定义了**标准 PASCAL**，这个标准是该语言各种实现的共同基础.

对于介绍一种语言来说,线性结构的书籍决不是理想的. 不过在作为教材使用时，我们建议读者还是按照用户手册的原有顺序阅读,阅读中要特别留意示范程序,以后再重读有困难的段落. 特别是当问题涉及到输入输出的约定时，可以参考第 12 章.

1) 此系本书第一、二部分的序言. ——译者注

用户手册的第 0 至 12 章以及整个报告，描述了标准 PASCAL. 实现人员应当把识别标准 PASCAL 的任务作为他的系统的基本要求；而程序员如果希望把他的程序可以从一个机器搬到另一个机器上，那么他只需要了解标准 PASCAL 所具有的特点. 当然，各种实现都可以提供附加的功能，但应当清楚地注明它们只是原有功能的扩充.

用户手册第 13 和 14 章是有关 PASCAL 在 Control Data 6000 计算机上实现的资料. 13 章描述被称为 PASCAL 6000 -3.4 语言的附加特色. 14 章着重说明这个编译程序在操作系统 SCOPE 3.4 下的使用情况.

这本手册是很多人努力的结果. 我们特别要感谢苏黎世瑞士工学院信息研究室的研究人员和 John Larmouth, Rudy Schild, Olivier Lecarme 及 Pierre Desjardins 给予的批评、建议和鼓励. 我们这个 PASCAL 实现是 Urs Ammann 在 Helmut Sandmayr 帮助下所作的工作，也正是由于这一工作才有可能而且有必要写这本手册，

K. 詹森　N. 沃思

于瑞士苏黎世瑞士工学院

1974 年 11 月

目　　录

序言 …… vii

第一部分　报告 …… 1

0. 引言 …… 1

1. 语言概述 …… 2

2. 表示法，术语和词汇表 …… 6

3. 标识符，数和串 …… 7

4. 常量定义 …… 9

5. 数据类型定义 …… 9

5.1. 简单类型 …… 9

5.2. 构造类型 …… 11

5.3. 指示字类型 …… 15

6. 变量的说明和标记 …… 16

6.1. 整体变量 …… 17

6.2. 成分变量 …… 17

6.3. 引用变量 …… 19

7. 表达式 …… 19

7.1. 运算符 …… 21

7.2. 函数命名符 …… 23

8. 语句 …… 23

8.1. 简单语句 …… 24

8.2. 构造语句 …… 26

9. 过程说明 …… 31

9.1. 标准过程 …… 35

10. 函数说明 …… 37

10.1. 标准函数 …… 38
11. 输入和输出 …… 40
11.1. 过程 read …… 41
11.2. 过程 readln …… 41
11.3. 过程 write …… 42
11.4. 过程 writeln …… 43
11.5. 其他过程 …… 43
12. 程序 …… 44
13. 实现和程序互换的标准 …… 45
索引 …… 46
第二部分 用户手册 …… 51
0. 引言 …… 51
1. 表示法和词汇表 …… 57
2. 数据的概念 …… 60
2.1. 布尔类型 …… 60
2.2. 整数类型 …… 61
2.3. 实数类型 …… 62
2.4. 字符类型 …… 63
3. 程序首部和说明部分 …… 65
3.1. 程序首部 …… 65
3.2. 标号说明部分 …… 65
3.3. 常量定义部分 …… 66
3.4. 类型定义部分 …… 67
3.5. 变量说明部分 …… 68
3.6. 过程与函数说明部分 …… 69
4. 动作的概念 …… 69
4.1. 赋值语句 …… 69
4.2. 复合语句 …… 71
4.3. 重复语句 …… 72
4.4. 条件语句 …… 77

4.5. 转向语句 …… 83
5. 纯量类型和子域类型 …… 85
5.1. 纯量类型 …… 85
5.2. 子域类型 …… 87
6. 构造类型概述，数组类型 …… 88
6.1. 数组类型 …… 88
7. 记录类型 …… 94
7.1. 开域语句 …… 101
8. 集合类型 …… 103
9. 文件类型 …… 109
9.1. 行文文件 …… 113
9.2. 标准文件 "input" 和 "output" …… 115
10. 指示字类型 …… 118
11. 过程和函数 …… 123
11.1. 过程 …… 124
11.2. 函数 …… 136
11.3. 附注 …… 141
12. 输入和输出 …… 141
12.1. 过程 read …… 143
12.2. 过程 write …… 144
13. PASCAL 6000-3.4 …… 146
13.1. 对 PASCAL 语言的扩充 …… 146
13.2. 前面各章中尚未确定的规定 …… 149
13.3. 限制 …… 155
13.4. 附加的预定义类型、过程和函数 …… 156
14. 如何使用 PASCAL 6000-3.4 系统
14.1. 控制语句（对 SCOPE 3.4） …… 158
14.2. 编译程序的任选 …… 159
14.3. 出错信息 …… 161
参考文献 …… 162

附录 A　标准过程和标准函数…… 163
附录 B　运算符一览表…… 166
附录 C　字定界符和标准标识符表…… 168
附录 D　语法…… 169
附录 E　出错号一览表…… 181
附录 F　程序设计实例…… 186
索引…… 191
第三部分　PASCAL 编译程序的设计 …… 198
0. 引言…… 198
1. 数据与程序的编排…… 200
1.1. 数据段和变量的编址 …… 200
1.2. 程序段 …… 202
1.3. 数据的表示 …… 204
1.4. 过程参数 …… 210
1.5. 代码优化 …… 211
2. 编译程序…… 212
2.1. 总的体制 …… 212
2.2. 编译时对象的描述 …… 213
2.3. 源行文的语法分析 …… 217
2.4. 性能与统计数据 …… 220
3. 编译程序设计技术…… 222
4. 编译复杂性和计算机总体结构的关系…… 225
5. 结论…… 227
参考文献…… 236
附录　语法图…… 237

第一部分　报　　告

N. 沃　思

0. 引　　言

研制 PASCAL 语言主要有两个目的. 第一,是想得到一种适宜于程序设计教学的语言，使以某些基本概念为基础的系统训练,可以通过该语言清楚、自然地得到反映.第二,是开发这种语言的实现方法，使这种实现在现有的各种计算机上既可靠又有效. 之所以迫切地希望有一种新的语言用于程序设计教学,是因为我对目前使用的大多数语言不满意,这些语言无论是其特性,还是其构成成分,几乎总是不能从逻辑上给予令人信服的解释，也几乎总是不给以系统的推理. 同时我还坚信,教给学生们用以表达他们思维的语言,会深深地影响他们思维和创造发明的习惯；而正是这些语言本身的混乱直接地影响着学生们程序设计的风格.

当然有许多理由表明，在引进一个新的程序设计语言时要持慎重态度，而对采用一个尚未获得广泛使用和接受的语言教授程序设计提出的异议,无疑也有一定的道理,至少从短期的商业利益上考虑是如此. 然而，造成这种具有深远教学影响的课题处于停滞不前的根本原因在于：基于广泛地被人们接受和使用来选择教学语言，而最为广泛地用于教学的语言又成为最为广泛使用的语言. 因而我认为，作一些努力来打破这种恶性循环是非常值得的.

当然，研制一个新的语言不应该只是为了新奇；至于现存的语言，只要它们能满足前面提到的准则，而且又不妨碍系统结构时，就应当成为发展的基础．正是在这个意义上，采用了 ALGOL 60 作为 PASCAL 的基础，因为它在教学上比其他的标准语言更能满足要求．于是，构造原则，实际上还有表达式的形式，都取自 ALGOL 60．但是，如果就采用 ALGOL 60 作为 PASCAL 的子集，我认为也不合适，因为它的某些结构原则，尤其是说明的结构，在用来表示 PASCAL 其他特点时，那又偏离了自然、方便这一目标．

较之 ALGOL 60，主要扩充在于数据构造工具方面，因为缺少这种工具是使 ALGOL 60 应用范围比较狭窄的主要原因．引进记录和文件结构将使得 PASCAL 有可能解决商业上的类型问题，或者至少可以在程序设计教程里成功地说明这些问题．

1. 语 言 概 述

一个算法或计算机程序由两个基本的部分组成，一个是对所要实现的**动作**的描述，一个是对这些动作所操作的**数据**的描述．动作是由通常所说的**语句**描述的，而数据则是由通常所说的**说明**和**定义**描述的．

数据是由**变量**的值表示的．语句中出现的每个变量都必须通过一个**变量说明**将其引进，变量说明把一个标识符和一个数据类型与那个变量联系起来．**变量的数据类型**本质上是定义了那个变量可以取的值之集合．PASCAL 中的数据类型可以直接在变量说明里描述，也可以用一个类型标识符来标记它．在后一种情况下，这个类型标识符必须用一个明显的**类型定义**来描述．

基本的数据类型是**纯量**类型. 它们的定义指示了一个值的有序集合，也就是引进一些标识符来代表这个集合中的每一个值. 除了这种可定义的纯量类型外，还有四种**标准基本类型**：**布尔**、**整数**、**字符**和**实数**. 它们的值除了布尔类型外，都不是用标识符而是用数和引文来标记的. 字符类型的值是特定的计算机设备上可用的字符集.

一个类型也可以定义为另一个纯量类型的**子域**，只要指出这个子域的最小值和最大值.

构造类型是由描述它们的成分的类型，并指明一种**构造方法**来定义的. 各种不同构造方法的区别在于从构造类型变量中选取成分时所采用的选择技术的不同. PASCAL 有四种可供选用的基本构造方法：数组结构、记录结构、集合结构和文件结构.

数组结构中的所有成分都属于同一类型. 成分是由数组选择器，即**可计算下标**选择的. 下标的类型要在该数组的类型定义中指明，而且必须是纯量类型；它通常是程序员定义的纯量类型，或是整数类型的子域. 给定一个下标类型的值，数组选择器就产生一个成分类型的值. 因此，每个数组变量可以看成是下标类型在成分类型上的一个映射. 为选择成分所需要的时间并不依赖于选择器(下标)的值. 数组结构因此而被称为是**随机存取结构**.

记录结构中的各成分（称为**字段**）不一定属于同一类型. 为了使得所选成分的类型能从程序行文上清楚地知道（毋需执行程序），记录选择器不是一个可计算的值，而是个标识符，它唯一地标记要选的那个成分. 这些成分标识符是在该记录的类型定义里说明的. 同样，存取一个选定的成分所需要的时间不依赖于选择器，因而记录结构也是一种随机存取结构.

一个记录类型可以被规定为由若干**变体**组成. 这就意味

着，不同的变量虽然属于同样的类型，但它们所采用的结构在某些方面可以有所不同．这些差别可以包括成分数目上的不同和成分类型上的不同．一个记录变量的当前值所设定的变体，可以用一个成分字段来指示，这个成分字段对所有变体都是公共的，并称为**标志字段**．通常，所有变体公共的部分将由包括标志字段在内的若干成分组成．

一个**集合结构**定义的值集合是它的基类型的势集，也就是基类型的值域的所有子集的集合．基类型必须是纯量类型，它通常总是程序员所定义的纯量类型，或是整数类型的子域．

一个**文件结构**是属于同一类型的成分的一个**序列**．这个序列定义了各成分的自然次序．在任何情况下，只有一个成分是直接可存取的，其他成分则要通过文件顺序前移才能存取．一个文件是由顺序地在它末端添加成分而生成的．因此，文件类型的定义并不决定成分的数目．

在显式说明中说明的变量被称为是**静态**的．这种说明使一个用于引用变量的标识符与它所引用的变量发生联系．可是，变量也可以借助于一个可执行语句来生成．这样一种**动态**生成则产生一个通常所说的**指示字**（代替显式标识符），它以后用于引用所生成的那个变量．这个指示字可以赋给其他的变量，即指示字类型的变量．每一个指示字变量可以取的值只能指向属于同一类型T的变量，并称它**受囿**于这个类型T．可是，指示字变量也可以取值 **nil**，它不指向任何变量．因为指示字变量也可以作为构造变量的成分，而构造变量其本身又可以是动态生成的，因此利用指示字就可以按照极其通用的方式表示有穷图．

最为基本的语句是**赋值语句**．它规定把一个新计算出的值赋给一个变量（或一个变量的成分）．该值是通过一个**表达**

式的计算而得到的．表达式由变量、常量、集合、运算符和函数组成．其中函数是在指定的量上进行运算并产生新值．变量、常量和函数，或是在程序里说明，或是标准实体．PASCAL定义了一个固定的运算符集，其每一个运算符都可以认为是描述一个从运算对象类型到结果类型的映射．运算符集分为下列几组：

1）**算术运算符**　加、减、取负、乘、除和计算余数．

2）**布尔运算符**　否定、析取(或)和合取(与)．

3）**集合运算符**　并、交和集合差．

4）**关系运算符**　相等、不等、次序、属于集合和集合包含．关系运算的结果属于**布尔**类型．

过程语句用于执行一个指定的过程(见下面)．赋值语句与过程语句是**构造语句**的成分或积木式部件．构造语句规定它们的成分语句要顺序地、有选择地、或重复地执行．语句的顺序执行由**复合语句**指明；条件的或选择的执行由**如果语句**和**分情形语句**指明；重复的执行由**直到语句**、**当语句**和**循环语句**指明．如果语句用于根据布尔表达式的值来执行某个语句，而分情形语句则根据选择器的值从许多语句中考虑选择．循环语句用于重复次数事先知道的情形，而直到语句和当语句则用于其他的重复情况．

一个语句可以给它一个名字(**标识符**)，并可由那个标识符来引用它．这种语句则被称为是**过程**，它的说明称为**过程说明**．这样一种说明还可以包含若干变量说明、类型定义和其他的过程说明．这样说明的变量、类型与过程只能在该过程内部引用，并因此称它们为**局部**于那个过程．它们的标识符仅在构成该过程说明的程序行文内有意义，并称这段行文是这些标识符的**作用域**．因为过程是可以说明为局部于其他过程的，所以作用域可以嵌套．在主程序里说明的实体，即不

是局部于某个过程的，被称为是**全程的**．一个过程有固定数目的参数，每个参数在过程内是由称之为**形式参数**的标识符标记．过程语句激活时，必须为每个形式参数指定一个实在量；实在量在过程语句内部可以借助于形式参数进行引用，这种量称为**实在参数**．共有四种参数：值参数、变量参数，过程参数和函数参数．在第一种情况下，实在参数是个表达式，它只求值一次，而这种形式参数是一个局部变量，表达式求值的结果在该过程(或函数)执行之前赋给这个局部变量．在变量参数的情况下，实在参数是个变量，形式参数则代表这个变量．如果实在参数带有下标，则它在过程（或函数）执行前计算．在过程参数或函数参数的情况下，实在参数是过程标识符或函数标识符．

函数的说明类似于过程．唯一的差别是函数要产生一个结果．结果的类型只限于纯量类型和指示字类型，并必须在这个函数的说明里指明．函数因此可以用作表达式的构成成分．为了消除副作用，在函数说明内应当避免对非局部变量赋值．

2. 表示法，术语和词汇表

按照传统的巴科斯-瑙尔（Backus-Naur）范式，语法构成元由括在尖括号〈与〉之间的中文词[1]标记．这些词也描述所标记的构成元的性质或含义，并用在相应的语义描述中．一个构成元若用元语言括号｛与｝括起来，则是指示该构成元允许重复[2]．符号〈空〉表示空的符号序列．

PASCAL 的基本词汇表由基本符号组成，它分为字母、数

1) 原文为“英文词”．——译者注

2) 也允许没有．——译者注

字和特殊符号三类.

〈字母〉::= A | B | C | D | E | F | G | H | I | J | K | L |
M | N | O | P | Q | R | S | T | U | V | W | X |
Y | Z | a | b | c | d | e | f | g | h | i | j |
k | l | m | n | o | p | q | r | s | t | u | v |
w | x | y | z

〈数字〉::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

〈特殊符号〉::= + | − | * | / | = | <> | < | > | <= | >
>= | (|) | [|] | { | } | := | . | , | ; | :
| ' | ↑ | **div** | **mod** | **nil** | **in** | **or** | **and** | **not** |
if | **then** | **else** | **case** | **of** | **repeat** | **until** |
while | **do** | **for** | **to** | **downto** | **begin** |
end | **with** | **goto** | **const** | **var** | **type** |
array | **record** | **set** | **file** | **function** |
procedure | **label** | **packed** | **program**

结构

{〈任何不包含"}"的符号序列〉}

可以插在任何两个标识符、数(参见3.)、或特殊符号之间. 它称为**注释**,程序行文中把它删除后并不改变程序的含义.符号"{"与"}"没有以其他方式出现在语言里[1]; 当它们在语法描述里出现时,和"|"与"::="一样,都是元语言符号. 符号对"(*"与"*)"用作"{"与"}"的同义词.

3. 标识符, 数和串

标识符用于标记常量、类型、变量、过程和函数[2]. 标识符

1) 除作为字符或串的字符外. ——译者注

2) 还有字段,形式参数和程序. ——译者注

和其所标记的对象之间的联系在它们合法的作用域内，即在说明它们的过程或函数内[1]（参见 9. 和 10.），必须是唯一的。

〈标识符〉::=〈字母〉{〈字母或数字〉}

〈字母或数字〉::=〈字母〉|〈数字〉

数采用通常的十进制表示法，它们是数据类型**整数**和**实数**（见 5.1.2.）的常量。标度因子前面的 E 表示“乘以 10 的…幂”。

〈数字序列〉::=〈数字〉{〈数字〉}

〈无符号整数〉::=〈数字序列〉

〈无符号实数〉::=〈无符号整数〉.〈数字序列〉|
〈无符号整数〉.〈数字序列〉E
〈标度因子〉|〈无符号整数〉E
〈标度因子〉

〈无符号数〉::=〈无符号整数〉|〈无符号实数〉

〈标度因子〉::=〈无符号整数〉|〈正负号〉
〈无符号整数〉

〈正负号〉::= + | —

例：

1　100　0.1　5E—3　87.35E+8

用引号括起来的字符序列称为**串**。由单个字符组成的串是标准类型**字符**（见 5.1.2.）的常量。由 n（＞1）个被括起来的字符所组成的串是类型（见 5.2.1）

packed array [1..n] **of** char

的常量。

注意：如果一个串要含有引号，那么这个引号必须连写两次。

1）“即”之后的描述对字段不适用，因为它的作用域不是过程与函数（见5.2.2节）。——译者注

〈串〉::='〈字符〉{〈字符〉}'

例:

'A' ';' ''''

'PASCAL' 'THIS IS A STRING'

4. 常 量 定 义

常量定义引入一个标识符作为某常量的同义词[1].

〈常量标识符〉::=〈标识符〉

〈常量〉::=〈无符号数〉|〈正负号〉〈无符号数〉|
〈常量标识符〉|〈正负号〉〈常量标识符〉|
〈串〉

〈常量定义〉::=〈标识符〉=〈常量〉

5. 数据类型定义

数据类型决定该类型的变量可以取的值的集合[2],并把一个标识符与该类型相联系[3].

〈类型〉::=〈简单类型〉|〈构造类型〉|〈指示字类型〉

〈类型定义〉::=〈标识符〉=〈类型〉

5.1. 简 单 类 型

〈简单类型〉::=〈纯量类型〉|〈子域类型〉|

1) 常量定义引入的常量标识符的类型由等号右边的常量决定. 前面冠有正负号的常量标识符,必须为整数(或其子域)与实数类型. ——译者注

2) 因而也决定了在该类型的量上可以进行的运算. ——译者注

3) 这里的"并"应改为"类型定义". 除指示字类型外,类型标识符的作用域不包括它本身的定义. ——译者注

〈类型标识符〉[1]

〈类型标识符〉::=〈标识符〉

5.1.1. 纯量类型

纯量类型借助于枚举标记值的标识符的方法来定义值的有序集合[2].

〈纯量类型〉::=(〈标识符〉{,〈标识符〉})

例:

(red, orange, yellow, green, blue)

(club, diamond, heart, spade)

(Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday)

应用于所有纯量类型(除实数类型外)[3]的函数是

succ 后继值(按枚举次序)

pred 前启值(按枚举次序).

5.1.2. 标准类型

下列类型在 PASCAL 中是标准的[4]:

integer (整数) 值是整数的子集,它由各个实现所定义.它的值是整数(见3.).

1) 类型标识符按其所表示的类型,应属于相应的类型.例如可以有简单类型标识符、纯量类型标识符、构造类型标识符,等等.故一概作为简单类型是不恰当的.这里应改为简单类型标识符,再加上相应的定义.而类型标识符的定义应列在5.中.——译者注

2) 这些标识符是作为常量标识符的另一种定义性出现.纯量类型这一定义并没有概括所有的纯量类型(包括标准的),具体地说没有包括类型 integer, real 与 char.其次,real 类型的使用有很大限制,它不能作为数组的下标类型、记录的标志字段的类型、集合类型的基类型、分情形语句的选择器类型、循环语句的控制变量的类型,也不可以用于定义子域类型.最好把它与其他的纯量类型从概念上加以区分.——译者注

3) 这里应加"的量"两字.此外,它们也能用于子域类型的量.——译者注

4) 为方便起见,下文中这些类型将用中文.——译者注

real （实数）	值是实数的子集，它依赖于具体实现。其值用实数表示（见 3.）。
Boolean（布尔）	值是由标识符 true（真）与 false（假）标记的逻辑真值。
char （字符）	值是由具体实现所决定的字符集合。它们由括在引号内的字符本身来标记。

5.1.3. 子域类型[1]

一个类型可以定义为另一纯量类型的子域，只要指定该子域的最小值和最大值。第一个常量指定下界，它不可大于上界。

〈子域类型〉::=〈常量〉..〈常量〉[2]

例：

```
1..100
-10..+10
Monday..Friday
```

5.2. 构 造 类 型

一个构造类型的特征在于它的成分类型和它的构造方法。其次，一个构造类型的定义可以包含一个指示，以表示所要求的数据表示方法。如果一个定义以符号 **packed** 作为前缀，那么它一般不影响程序的含义（限制见 8.1.2.）[3]；它只是

1）子域类型除在 5.1. 节中作为不同于纯量类型的类型外，在其他章节中大多都未单独提及，均算作纯量类型。——译者注

2）这两个常量都必须属于除实数类型外的简单类型。——译者注

3）这里未说明前缀 **packed** 对构造类型影响的层次深度，大概只影响以它为前缀的那层构造类型的表示；也未说明加了 **packed** 后对类型一致性问题的影响。——译者注

暗示编译程序应当节省存贮，甚至以损失一些存取效率作为代价，即使这样做有可能使得存取该结构成分所需要的代码加长.

〈构造类型〉::=〈非紧缩构造类型〉|
packed〈非紧缩构造类型〉

〈非紧缩构造类型〉::=〈数组类型〉|〈记录类型〉|
〈集合类型〉|〈文件类型〉

5.2.1. 数组类型

数组类型是一个由固定数目的成分所组成的结构，所有成分都属于同一类型，并称它为成**分类型**. 数组的元素由下标指定，下标的值属于所谓的**下标类型**. 数组类型的定义既规定了它的成分类型，也规定了它的下标类型.

〈数组类型〉::=**array** [〈下标类型〉{,〈下标类型〉}]
of〈成分类型〉

〈下标类型〉::=〈简单类型〉[1]

〈成分类型〉::=〈类型〉

如果规定了 n 个下标类型，那么该数组类型称为是 **n-维**的，它的成分要由 n 个下标来指定.

例:

```
array [1..100]of real
array [1..10, 1..20] of 0..99
array [Boolean] of color
```

5.2.2 记录类型

记录类型是个由固定数目的成分所组成的结构，但各成分的类型可以不同. 记录类型的定义对它每个称为**字段**的成

1) 除实数类型外. ——译者注

分，规定其类型和一个用以标记它的标识符[1]．这些所谓的**字段标识符**的作用域是记录定义本身，并且它们也可以在与该类型的记录变量有关的字段命名符(参见 6.2.) 内使用[2]．

一个记录类型可以有若干**变体**，在此情况下可以指定某个字段为**标志字**段，它的值指示现时哪个变体为记录变量所采用．每个变体结构都由一个分情形标号所标识，它是标志字段的类型的一个常量[3]．

〈记录类型〉::=**record**〈字段表〉**end**

〈字 段 表〉::=〈固定部分〉|〈固定部分〉;
〈变体部分〉|〈变体部分〉

〈固定部分〉::=〈记录段〉{;〈记录段〉}

〈记 录 段〉::=〈字段标识符〉{,〈字段标识符〉}
:〈类型〉|〈空〉

〈变体部分〉::=**case**〈标志字段〉〈类型标识符〉**of**
〈变体〉{;〈变体〉}[4]

〈变体〉::=〈分情形标号表〉:(〈字段表〉)|〈空〉

〈分情形标号表〉::=〈分情形标号〉{,〈分情形标号〉}

〈分情形标号〉::=〈常量〉

〈标志字段〉::=〈标识符〉:|〈空〉

例:

```
record day: 1..31;
```

1) 这是字段标识符的定义性出现．——译者注

2) 也参见 8.2.4. ——译者注

3) 对标志字段非空的记录，其某个分情形标号等于标志字段当前值的变体是该记录现时所选用的记录．当标志字段的值指向其他变体时，暗示所选变体的变更．但对于没有标志字段的记录，选用变体的所指与变更是由对某变体的字段的引用来暗示．所选变体变更时，老变体的值不复存在，新变体的值无定义．——译者注

4) 这里的"标志字段"从它的语法定义看最好加上"部分"两字，而类型标识符应该为非实数类型的简单类型标识符．——译者注

```
        month: 1..12;
        year: integer
    end
    record name, firstname: alfa;
        age: 0..99;
        married: Boolean
    end
    record x,y: real;
        area: real;
      case s: shape of
      triangle: (side:real;
                 inclination, angle1, angle2: angle);
      rectangle: (side1, side2: real;
                  skew, angle3: angle);
      circle:    (diameter: real)
    end
```

5.2.3. 集合类型

集合类型定义的值域是它所谓的**基类型**的势集．基类型不可为构造类型[1]．可应用于所有集合类型的运算符是

+	并
−	集合差
*	交
in	属于关系

集合差 x − y 被定义为 x 中所有不属于 y 的元素的集合．

〈集合类型〉::= **set of** 〈基类型〉

〈基类型〉::=〈简单类型〉

5.2.4. 文件类型

文件类型指明一个由成分序列所组成的结构，其所有成

1) 也不可以为实数类型．——译者注

分都属于同一类型．成分的数目称为该文件的**长度**，它不是由文件类型的定义所决定的．零个成分所组成的文件称为**空文件**．

〈文件类型〉::= **file of** 〈类型〉

成分类型是字符的文件称为**行文文件**；但这是一种特殊情况，因为它的成分的值域必须看成是已扩充了一个标记符号，用以表示一行结束．这个符号使行文文件分成行．类型**text**（**行文**）是下面预定义的标准类型：

type text = **file of** char

5.3. 指示字类型

在一程序中说明的变量(见6.)可借助于它们的标识符进行存取．这些变量在它们所局部于的那个过程[1]（作用域）的整个执行期间都存在，这些**变量**因此被称为是**静态**的(或静态分配的)．与此相对，变量也可以动态地生成，即与该程序的结构没有任何关系．这些**动态**的变量是由标准过程 **new**（见9.1.2.）生成的；因为它们不在显式变量说明里出现，所以不能用名字来引用它们．代之以存取是由通常所说的**指示字**值实现的，这个值由该动态变量的生成提供．于是一个指示字类型是由值的无界集合所组成，该集合的元素均指向同一类型的元素．除了赋值和检查相等外，在指示字上没有定义任何其他的运算．

指示字值 **nil** 属于任何指示字类型；它根本不指向任何元素．

〈指示字类型〉::= ↑〈类型标识符〉

1）或函数．——译者注

类型定义[1]的例子：

```
color   = (red, yellow, green, blue)
sex     = (male, female)
text    = file of char
shape   = (triangle, rectangle, circle)
card    = array [1..80] of char
alfa    = packed array [1..10] of char
complex = record re, im: real end
person  = record name, firstname: alfa;
                 age: integer;
                 married: Boolean;
                 father, child, sibling: ↑person;
            case s: sex of
              male: (enlisted, bold: Boolean);
              female: (pregnant: Boolean;
                       size: array [1..3] of integer)
            end
```

6. 变量的说明和标记

变量说明是由一个标识符表后面加上它们的类型组成，这些标识符标记新的变量。

〈变量说明〉::=〈标识符〉{,〈标识符〉}:〈类型〉

对于成分的类型为T的文件变量f，它的说明都蕴涵一个类型T的所谓**缓冲器变量**的附加说明．这个缓冲器变量记为

1) 类型一致性问题，例如：对两个定义相同的类型，起名的与不起名的，或起不同名字的；子域类型和作为其母类型的纯量类型；带 **packed** 的与不带的；常量(包括集合常量)类型的确定；集合，串的比较；数组下标的设定；实参与形参的替换；赋值；循环语句的控制变量与循环上下界，等等．PASCAL 语言对此没有作明确的定义或规定．报告中虽使用了三个词"同一的(same)"、"全同的(identical)"和"符合于(correspond with)"，但看不出是作为三类一致性问题提出的(如果这样认为，可能也不一定恰当)，也没有给这三个词的精确定义．——译者注

f↑；它在文件生成时用于给文件添加成分，在检验时用于存取该文件（见6.2.3.和9.1.1.）。

例：

```
x, y, z: real
u, v: complex
i, j: integer
k: 0..9
p,q: Boolean
operator: (plus, minus, times)
a: array [0..63] of real
b: array [color, Boolean] of complex
c: color
f: file of char
hue1, hue2: set of color
p1,p2: ↑person
```

变量的标记是指称一个整体变量、一个变量的一个成分、或由一指示字（见5.3.）所引用的一个量。下面各章的例子中的变量，都假定是象上面那样说明的。

〈变量〉::=〈整体变量〉|〈成分变量〉|〈引用变量〉

6.1. 整 体 变 量

整体变量是用它的标识符标记的。

〈整体变量〉::=〈变量标识符〉

〈变量标识符〉::=〈标识符〉

6.2. 成 分 变 量

变量的成分是由该变量后面跟一个规定该成分的选择器来标记的。选择器的形式依赖于变量的构造类型。

〈成分变量〉::=〈下标变量〉|〈字段命名符〉|
〈文件缓冲器〉

6.2.1. 下标变量

n 维数组变量的成分是由该变量后面跟以 n 个下标表达式来标记的.

〈下标变量〉::=〈数组变量〉[〈表达式〉{,〈表达式〉}]

〈数组变量〉::=〈变量〉

下标表达式的类型必须符合于该数组类型的定义中所说明的下标类型.

例:

```
a [12]
a [i + j]
b [red, true]
```

6.2.2. 字段命名符

记录变量的成分是由该变量后面跟有成分的字段标识符来标记的.

〈字段命名符〉::=〈记录变量〉.〈字段标识符〉

〈记录变量〉::=〈变量〉

〈字段标识符〉::=〈标识符〉

例:

```
u.re
b [red, true].im
p 2 ↑.size
```

6.2.3. 文件缓冲器

一个文件变量在任何时刻,只有当前文件位置(读/写头)

所决定的那一个成分才是可存取的. 这个成分称为当前文件成分,并由该文件的**缓冲器变量**所表示.

〈文件缓冲器〉::=〈文件变量〉↑

〈文件变量〉::=〈变量〉

6.3. 引 用 变 量

〈引用变量〉::=〈指示字变量〉↑

〈指示字变量〉::=〈变量〉

如果 p 是受囿于类型 T 的指示字变量，那么 p 是标记这个变量和它的指示字值，而 p↑ 则标记由 p 所引用的类型为 T 的变量.

例:

```
p 1 ↑.father
p 1 ↑.sibling↑.child
```

7. 表 达 式

表达式是表示计算规则的结构，以便可以运用运算符来得到变量的值或产生新的值. 表这式由运算符和运算对象即变量、常量与函数组成.

表达式的形成规则指明了运算符的**优先顺序**，它依据四类运算符. 运算符 **not** 的优先顺序最高,接着是所谓乘型运算符,再下面是所谓加型运算符,最后的即最低优先顺序的是关系运算符. 具有同样优先顺序的运算符的序列则自左向右依次执行. 优先顺序的规则反映在下面的语法里:

〈无符号常量〉::=〈无符号数〉|〈串〉|〈常量标识符〉|**nil**

〈因子〉::=〈变量〉|〈无符号常量〉|〈函数命名符〉|

〈集合〉|(〈表达式〉)|**not** 〈因子〉

〈集合〉::=[〈元素表〉]

〈元素表〉::=〈元素〉{,〈元素〉}|〈空〉

〈元素〉::=〈表达式〉|〈表达式〉..〈表达式〉

〈项〉::=〈因子〉|〈项〉〈乘型运算符〉〈因子〉

〈简单表达式〉::=〈项〉|〈简单表达式〉
〈加型运算符〉〈项〉|
〈加型运算符〉〈项〉

〈表达式〉::=〈简单表达式〉|〈简单表达式〉
〈关系运算符〉〈简单表达式〉

用作集合元素的各表达式必须属于同一类型，即是该集合的基类型. [] 表示空集合，[x..y] 表示区间 x..y 中所有值的集合.

例:

因子: x
15
(x + y + z)
sin(x + y)
[red, c, green]
[1, 5, 10..19, 23]
not P

项: x*y
i/(1 - i)
P or q
(x <= y) and (y < z)

简单表达式: x + y
-x
huel + hue2

```
                i*j + I
表达式:         x = 1.5
                p< =q
                (i<j) = (j<k)
                c. in huel
```

7.1. 运 算 符

如果算术运算符加、减和乘的两个运算对象均属于整数类型(或它的子域),那么结果也属于整数类型. 如果有一个运算对象属于实数类型,那么结果就属于实数类型.

7.1.1. 运算符 not

运算符 **not** 标记它的布尔运算对象的否定.

7.1.2. 乘型运算符

〈乘型运算符〉::= * | / | **div** | **mod** | **and**

运 算 符	运 算	运算对象的类型	结果的类型
*	乘 集 合 交	实数、整数 任何集合类型T	实数、整数 T
/	除	实数、整数	实数
div	截 尾 除	整数	整数
mod	取 模	整数	整数
a d	逻辑"与"	布尔	布尔

7.1.3. 加型运算符

〈加型运算符〉::= + | − | **or**

运 算 符	运 算	运算对象的类型	结果的类型
+	加 集 合 并	整数、实数 任何集合类型T	整数、实数 T
−	减 集 合 差	整数 实数 任何集合类型T	整数、实数 T
or	逻辑"或"	布尔	布尔

当运算符用于只有一个运算对象时，"−"表示取负运算，"+"表示恒等运算.

7.1.4. 关系运算符

〈关系运算符〉::=＝|＜＞|＜|＜＝|＞＝|＞|in

运 算 符	运 算 对 象 的 类 型	结果的类型
= < >	任何纯量类型或子域类型[1]	布尔
< >		
<=>=		
in	任何纯量类型[2]或子域类型和它相应的集合类型	布尔

1) 对指示字类型也允许＝与＜＞运算. ——译者注
2) 除实数类型外. ——译者注

注意：所有纯量类型都定义了值的有序集合.

运算符 ＜ ＞、＜＝ 和 ＞＝，分别代表不等、小于等于和大于等于.

运算符 ＜＝和＞＝ 也可以用于比较集合类型的值，它表示集合的包含关系.

如果 p 与 q 是布尔表达式，那么 p = q 表示它们的等价关系，而 p<=q 则表示 p 蕴涵 q. (注意：false<true)

关系运算符=、<>、<、<=、>和>=，也可以用于比较其成分的类型为字符的(紧缩)数组(即串)，它表示以基本字符集的排序序列为基础的字典次序。

7.2. 函数命名符

函数命名符指明一个函数的激活．它由指称该函数的标识符和一个实在参数表组成．这些参数是变量、表达式、过程和函数，并替代对应的形式参数(参见 8.1.2.，9. 与 10.)．

〈函数命名符〉::=〈函数标识符〉|〈函数标识符〉
(〈实在参数〉{,〈实在参数〉})

〈函数标识符〉::=〈标识符〉

例：

```
Sum (a, 100)
GCD (147, k)
sin (x + y)
eof (f)
ord (f↑)
```

8. 语　　句

语句标记算法的动作，并称为是**可执行的**．它们可以前缀一个能被转向语句引用的标号．

〈语句〉::=〈无标号语句〉|〈标号〉:〈无标号语句〉

〈无标号语句〉::=〈简单语句〉|〈构造语句〉

〈标号〉::=〈无符号整数〉

8.1. 简 单 语 句

简单语句是一种它的任何部分都不构成另一语句的语句. 空语句不包含任何符号,也不标记任何动作.

〈简单语句〉::=〈赋值语句〉|〈过程语句〉|〈转向语句〉|
〈空语句〉

〈空语句〉::=〈空〉

8.1.1. 赋值语句

赋值语句用于以一个表达式所指明的新值代替一个变量[1]的当前值.

〈赋值语句〉::=〈变量〉:=〈表达式〉|
〈函数标识符〉:=〈表达式〉

变量(或函数)和表达式必须属于全同的类型,但允许有下列例外:

1) 变量的类型是实数,而表达式的类型是整数或它的子域.

2) 表达式的类型是变量的类型的子域,或变量的类型是表达式的类型的子域.

例:

```
x := y + z
p := (1<=i)and (i<100)
i := sqr(k) - (i*j)
huel := [blue, succ (c)]
```

8.1.2. 过程语句

过程语句用于执行由过程标识符所标记的过程. 过程语

1) 或函数. ——译者注

句可以包含一个**实在参数**表，这些实在参数替代在该过程的说明中所定义的并与它们相对应的**形式参数**(参见 9.)．这种对应关系是由参数在实在参数表和形式参数表中的相应位置确定的．有四类参数：值参数、变量参数、过程参数(实在参数是个过程标识符)和函数参数(实在参数是个函数标识符)．

在**值参数**的情况下，实在参数必须是一个表达式(变量是它的简单情形)．对应的形式参数代表被调用过程的一个局部变量，作为实在参数的表达式在该过程激活时以它的当前值赋给这个变量．在**变量参数**的情况下，实在参数必须是个变量，对应的形式参数在该过程整个执行期间都代表这个实在变量．如果这个变量是一个数组的成分，那么它的下标在该过程调用时计算．每当参数表示过程的结果时，必须使用变量参数[1]．

紧缩结构的成分不可以作为实在变量参数．

〈过程语句〉::＝〈过程标识符〉|〈过程标识符〉
(〈实在参数〉{,〈实在参数〉})

〈过程标识符〉::＝〈标识符〉

〈实在参数〉::＝〈表达式〉|〈变量〉|〈过程标识符〉|
〈函数标识符〉

例：

```
next
Transpose (a, n, m)
Bisect(fct, −1.0, +1.0, x)
```

8.1.3. 转向语句

转向语句用于指示进一步的加工将在程序行文的另一部

1) 这里没有说明形式参数和实在参数之间的类型关系，也没有说明形式过程(函数)的参数和实在过程(函数)的参数之间的关系．——译者注

分，即在所指定的标号的位置上继续.

〈转向语句〉::=**goto**〈标号〉

对标号的应用有下列限制[1]:

1）标号的作用域是它的定义所在的过程，因此转进一个过程是不可能的.

2）每个标号必须在它标记的语句所在过程的首部的标号说明里指明.

8.2. 构造语句

构造语句是由若干其他语句组成的结构，这些成分语句必须是顺序地执行(复合语句)，条件地执行(条件语句)，或重复地执行(重复语句)[2].

〈构造语句〉::=〈复合语句〉|〈条件语句〉|〈重复语句〉|
〈开域语句〉

8.2.1. 复合语句

复合语句指明它的成分语句要按它们书写的顺序执行. 符号 **begin** 与 **end** 起语句括号的作用.

〈复合语句〉::=**begin**〈语句〉{;〈语句〉}**end**

例:

begin z := x; x := y; y := z **end**

8.2.2. 条件语句

条件语句从它的成分语句中选择执行其中一个语句.

1）没有规定是否允许转入构造语句. 看来不一定允许，例如循环语句. ——译者注

2）没有概括进"开域语句"之情形，拟加上"或在扩展的作用域内执行（开域语句)". ——译者注

〈条件语句〉::=〈如果语句〉|〈分情形语句〉

8.2.2.1 如果语句

如果语句指明一个语句被执行仅当某个条件（布尔表达式)为真. 如果为假，那么或者不执行任何语句，或者执行跟在符号 **else** 后面的语句.

〈如果语句〉::=**if**〈表达式〉**then**〈语句〉|
if〈表达式〉**then**〈语句〉**else**〈语句〉

符号 **if** 和 **then** 之间的表达式必须属于布尔类型.

注意：由结构

if〈表达式1〉**then if**〈表达式 2〉
then〈语句 1〉**else**〈语句 2〉

引起的语法上二义性问题的解决办法是把它解释为等价于

if〈表达式 1〉**then**
begin if〈表达式 2〉**then**〈语句 1〉
else〈语句 2〉
end

例：

if x<1.5 **then** z: = x + y **else** z:=1.5

if p1 < > **nil then** p1 := p1↑. father

8.2.2.2. 分情形语句

分情形语句是由一个表达式（选择器）和一个语句表组成，每个语句均有一个选择器类型的常量[1]作为它的标号. 分情形语句指明其标号等于选择器当前值的一个语句被执行.

〈分情形语句〉:: = **case**〈表达式〉**of**〈分情形表元素〉

1）一个常量的类型怎样才算是表达式的类型，未精确定义. 此外，该类型应限为非实数类型的简单类型. ——译者注

{;〈分情形表元素〉} **end**

〈分情形表元素〉::=〈分情形标号表〉:〈语句〉|〈空〉

〈分情形标号表〉::〈分情形标号〉{,〈分情形标号〉}

例:

```
case operator of
    plus:   x := x + y;
    minus:  x := x - y;
    times:  x := x*y
end
```

```
case i of
    1: x := sin(x);
    2: x := cos(x);
    3: x := exp(x);
    4: x := ln(x)
end
```

8.2.3. 重复语句

重复语句指明某些语句要重复地执行. 如果重复次数事先知道,即在重复开始前知道,那么循环语句是宜于表达这种情形的结构;不然的话,应该使用当语句或直到语句.

〈重复语句〉::=〈当语句〉|〈直到语句〉|〈循环语句〉

8.2.3.1. 当语句

〈当语句〉::= **while** 〈表达式〉 **do** 〈语句〉

控制重复的表达式必须属于布尔类型. **do** 后面的语句被重复地执行,直至表达式的值为假. 如果它的值一开始就是假,则那个语句根本不执行. 当语句

while B **do** S

等价于

```
if B then
    begin S;
        while B do S
    end
```

例:

```
while a [i] <> x do i := i + 1

while i>0 do
begin if odd(i) then z := z*x;
        i := i div 2;
        x := sqr(x)
end

while not eof (f) do
begin p (f↑); get (f)
end
```

8.2.3.2. 直到语句

〈直到语句〉::=**repeat**〈语句〉{;〈语句〉}
until〈表达式〉

控制重复的表达式必须属于布尔类型. 在符号 **repeat** 与 **until** 之间的语句序列被重复地执行(至少一次),直到表达式的值为真. 直到语句

```
repeat S until B
```

等价于

```
begin S;
    if not B then
        repeat S until B
end
```

例:

```
repeat k := i mod j;
        i := j;
        j := k
until j = 0

repeat P(f↑); get(f)
until eof(f)
```

8.2.3.3. 循环语句

循环语句指示一个语句在一个值级数的各值逐个赋给某个变量时被重复地执行. 这个变量称为该循环语句的**控制变量**[1].

〈循环语句〉::= **for** 〈控制变量〉:=〈循环表〉 **do** 〈语句〉

〈循环表〉::=〈初值〉 **to** 〈终值〉|〈初值〉 **downto** 〈终值〉

〈控制变量〉::=〈标识符〉

〈初值〉::=〈表达式〉

〈终值〉::=〈表达式〉

控制变量、初值与终值必须属于同样的纯量类型(或其子域),并且决不可被重复的语句所变更. 它们不能属于实数类型.

形式为

for v:=e1 **to** e2 **do** S

的循环语句等价于语句序列

v:=e 1; S; v:=succ(v); S; ···; v:=e 2; S

而形式为

for v:=e1 **downto** e2 **do** S

的循环语句等价于语句序列

v := e1; S; v: = pred (S); S; ···; v: = e2; S

例:

```
for i := 2 to 63 do if a[i] > max then max := a [i]

for i := 1 to n do
for j := 1 to n do
begin x := 0;
    for k := 1 to n do x := x + A[i, k] * B[k, j];
```

1) 控制变量在这里不是定义性出现. ——译者注

```
        C[i, j] := x
    end

    for c := red to blue do Q(c)
```

8.2.4. 开域语句

〈开域语句〉:: = **with** 〈记录变量表〉 **do** 〈语句〉

〈记录变量表〉:: = 〈记录变量〉{,〈记录变量〉}

在开域语句的成分语句内，由开域子句所指明的记录变量的成分（字段）可以仅用它们的字段标识符来标记，也就是说它们前面毋须带整体变量的标记. 开域子句有效地打开了所指明的记录变量的字段标识符所在的作用域，使得字段标识符可以象变量标识符那样地出现[1].

例:

```
with date do
if month = 12 then
    begin month := 1; year := year + 1
    end
else month := month + 1
```

等价于

```
if date.month = 12 then
    begin date.month := 1; date.year := date.year + 1
    end
else date.month := date.month + 1
```

在所限定的语句内，不能对记录变量表的任何元素赋值. 但是，对这些变量的成分可以赋值.

9. 过 程 说 明

过程说明用于定义一部分程序，将其与一标识符相联系，

1) 即一个记录变量在记录变量表中的出现，是它的字段标识符作为变量标识符对该"开域语句"的定义性出现. ——译者注

使得这部分程序可通过一个过程语句来激活.

〈过程说明〉:: =〈过程首部〉〈分程序〉[1]

〈分程序〉:: =〈标号说明部分〉〈常量定义部分〉
〈类型定义部分〉〈变量说明部分〉
〈过程与函数说明部分〉〈语句部分〉

过程首部指明作为该过程名字的标识符和若干形式参数标识符(若有的话). 这些参数是值参数、变量参数、过程参数或函数参数 (也参见 8.1.2.). 过程与函数作为其他过程与函数的参数时,只可以有值参数.

〈过程首部〉:: = **procedure**〈标识符〉;|
procedure〈标识符〉
(〈形式参数段〉{;〈形式参数段〉});

〈形式参数段〉:: =〈参数组〉| **var**〈参数组〉|
function〈参数组〉|
procedure〈标识符〉{,〈标识符〉}[2]

〈参数组〉:: =〈标识符〉{,〈标识符〉}:〈类型标识符〉

没有前缀区分符的参数组蕴涵它的各元素是值参数.

标号说明部分指明了所有在语句部分中标记语句用的标号.

〈标号说明部分〉:: =〈空〉| **label**〈标号〉{,〈标号〉};

常量定义部分包含所有局部于该过程的常量同义词的定义.

〈常量定义部分〉:: =〈空〉| **const**〈常量定义〉

1) 这里未提向前引用方式. "用户手册"指出只有存在向前引用时才允许先调用后说明. 但向前引用对于互相调用的情形是必不可少的. 当然, 报告没有象手册那样明确要求量必须先说明后引用. ——译者注

2) 函数参数和过程参数的语法未规定相应的实在函数和实在过程的参数要求, 故本定义的第三、四两项分别改为函数首部与过程首部较好, 当然这些首部只能有值参数. ——译者注

{;〈常量定义〉};

类型定义部分包含所有局部于该过程说明的类型定义.

〈类型定义部分〉::=〈空〉| **type**〈类型定义〉

{;〈类型定义〉};

变量说明部分包含所有局部于该过程说明的变量说明.

〈变量说明部分〉::=〈空〉| **var**〈变量说明〉

{;〈变量说明〉};

过程与函数说明部分包含所有局部于该过程说明的过程说明与函数说明.

〈过程与函数说明部分〉::={〈过程或函数说明〉;}

〈过程或函数说明〉::=〈过程说明〉|〈函数说明〉

语句部分指明该过程被过程语句激活时所要执行的算法动作.

〈语句部分〉::=〈复合语句〉

在形式参数部分、常量定义部分、类型定义部分、变量说明部分和过程与函数说明部分里引进的所有标识符都局部于该过程说明，并把后者称为是这些标识符的**作用域**. 它们在其作用域外面是不知道的. 就局部变量而言，它们的值在语句部分一开始无定义[1].

过程标识符在它的说明内的一个过程语句里的使用，蕴涵着该过程的递归执行.

过程说明的例子:

```
procedure readinteger (var f:text; var x: integer);
```

1) PASCAL 的类型、常量(包括纯量类型定义引入的)、变量、函数和过程(标识符)的作用域规则和 ALGOL60 相似，这里叙述不严格，还应除去同一标识符在内部嵌套层次中重新定义所在的过程说明与函数说明（参见 9.1. 及手册 3.5.). 标号的作用域规则同上，但它在标号说明部分中及语句前的出现，哪个是定义性出现，PASCAL 未明确规定(似乎是后者). 字段标识符的作用域规则参见 5.2.2. 节及8.2.4. 的译者注. ——译者注

```
var i, j: integer;
begin while f↑ = ' ' do get (f); i :=0;
    while f↑ in ['0'..'9'] do
        begin j:= ord (f↑ ) - ord('0');
            i := 10 * i + j;
            get(f)
        end;
    x := i
end

procedure Bisect (function f: real; a,b: real; var z: real);
var m: real;
begin {假定 f(a) <0 且 f(b) > 0 }
    while abs (a - b) > 1E - 10 * abs(a) do
    begin m := (a + b)/2.0;
        if f(m) < 0 then a := m else b := m
    end;
    z := m
end

procedure GCD(m, n: integer; var x, y, z: integer);
var a1, a2, b1, b2, c, d, q, r: integer; {m>=0, n>0}
begin {m 与 n 的最大公因子 x,扩充的欧几里得算法}
    a1 := 0; a2 := 1; b1 := 1; b2 := 0;
    c := m; d := n;
    while d < > 0 do
    begin {a1*m + b1*n=d, a2*m + b2*n = c,
            gcd(c, d) = gcd(m, n)}
        q := c div d; r := c mod d;
        a2 := a2 - q*a1; b2 := b2 - q*b1;
        c := d; d := r;
        r := a1; a1 := a2; a2 := r;
        r := b1; b1 := b2; b2 := r
    end
    x := c; y := a2; z := b2
    { x=gcd (m, n) = y*m + z*n }
end
```

9.1. 标 准 过 程

标准过程在 PASCAL 的每个实现里都被假定是预先定义的. 任何实现都可以添加其他预先说明的过程，以此作为自己的特色. 因为它们与所有其他标准量一样，都假定是在围绕该程序的作用域里说明的，所以并不会由于程序内某个说明对同一标识符进行重新定义而发生冲突. 下面列出了标准过程,并给以解释.

9.1.1. 文件处理的过程

put(f)　把缓冲器变量 f↑ 的值添加到文件 f 上. 仅当它执行前谓词 eof(f) 为真时,它的作用才有定义. eof(f) 仍为真，f↑ 的值变成无定义.

get (f)　把当前文件位置（读/写头）移到下一成分上，并把这个成分的值赋给缓冲器变量 f↑. 如果并不存在下个成分,那么 eof(f) 变成真，f↑ 的值没有定义. 仅当它执行前 eof(f) 为假时，get(f) 的作用才有定义. (见 10.1.2.)

reset(f)　把当前文件位置恢复到它的开始，并把 f 的第一个元素的值赋给缓冲器变量 f↑. 若 f 不空,则 eof (f) 变成假; 否则 f↑ 无定义，eof(f) 仍为真.

rewrite (f)　擦去 f 的当前值，使得可以生成新文件. eof(f) 变成真.

关于过程 read，write，readln，writeln 与 page 见 11 章.

9.1.2. 动态分配的过程

new(p)　　分配一个新变量 v，并把对 v 的指示字赋给指示字变量 p[1]. 如果 v 的类型是带变体的记录类型，那么形式

new(p,t1,··· tn)　可用于分配标志字段的值为 t1,···, tn 的变体的变量. 标志字段的值必须按照它们说明的次序连续地列出，并且在执行期间不可以改变.

dispose(p)　　指示由变量 p↑ 占用的存贮不再需要. 如果该变量是使用 new 的第二种形式分配的，则必须使用带有同样标志字段值的

dispose(p, t1,···, tn),　它指示这个变体所占的存贮不再需要.

9.1.3. 数据转换的过程

设变量 a 和 z 说明为

```
a: array [m··n] of T
z: packed array [u··v] of T
```

其中 n − m>=v − u. 那么语句 pack(a, i, z) 的作用是

```
for j := u to v do z[j] := a[j − u + i],
```

而语句 unpack (z, a, i) 的作用是

```
for j := u to v do a[j − u + i] := z[j],
```

其中 j 是标记不在程序其他地方出现的辅助变量.

1) 若指示字变量 P 指示的类型为 T，那么新变量属于类型 T. ——译者注

10. 函数说明

函数说明用于定义计算纯量值或指示字值的部分程序。函数是由函数命名符的求值而激活的（参见7.2.），函数命名符是表达式的构成部分。

〈函数命名符〉::=〈函数首部〉〈分程序〉[1]

函数首部指明作为该函数名字的标识符、该函数的形式参数和该函数的类型。

〈函数首部〉::=**function**〈标识符〉:〈结果类型〉;|
function〈标识符〉(〈形式参数段〉
{;〈形式参数段〉}):〈结果类型〉;

〈结果类型〉::=〈类型标识符〉

函数的类型必须是纯量类型、子域类型、或指示字类型。在函数说明的内部，必须至少存在一个对它的函数标识符进行赋值的赋值语句。这种赋值确定该函数的结果[2]。函数标识符在它的说明内的一个函数命名符里的出现，蕴涵着该函数的递归执行。

例：

```
function Sqrt(x: real): real;
var x0, x1: real;
begin x1 := x; {x>1, 牛顿法}
    repeat x0 := x1; x1 := (x0 + x/x0)*0.5
    until abs(x1 - x0) <eps*x1;
    Sqrt := x0
end

functionMax (a:vector; n: integer): real;
```

1) 参见9节的译者注。——译者注

2) 这里也应有象过程说明中那样的作用域说明。——译者注

```
var x: real; i: integer;
begin x := a[1];
    for i := 2 to n do
    begin {x=max (a[1], ···, a[i - 1]}
        if x < a[i] then x := a[i]
    end;
    {x=max(a[1] ,··· ,a [n])}
    Max := x
end


function GCD(m,n: integer): integer;
begin if n = 0 then GCD := m else GCD := GCD (n, m mod n)
end


function Power(x: real; y: integer): real; {y>=0}
var w, z: real; i: integer;
begin w :=x ; z :=1 ; i :=y ;
    while i>0 do
    begin {z* (w**i) = x**y}
        if odd (i) then z := z*w;
        i := i div 2;
        w := sqr(w)
    end;
    {z=x**y}
    Power : =z
end
```

10.1. 标 准 函 数

标准函数在 PASCAL 的每个实现里都被假定是预先定义的. 任何实现都可以添加其他预先说明的函数，以此作为自己的特色 (也参见 9.1.).

下面列出了标准函数，并给以解释.

10.1.1. 算术函数

abs(x)	计算 x 的绝对值. x 的类型必须是**实数**或**整数**,结果的类型是 x 的类型.
sqr(x)	计算 x**2. x 的类型必须是**实数或整数**,结果的类型是 x 的类型.
sin (x) cos (x) exp(x) ln(x) sqrt (x) arctan (x)	x 的类型必须是**实数**或**整数**,结果的类型是**实数**.

10.1.2. 谓词

odd(x)	x 的类型必须是**整数**;若 x 是奇数,则结果为真,否则为假.
eof(f)	eof(f) 指示文件 f 是否处在文件结束状态.
eoln(f)	指示行文文件中一行的结束(见 11 章).

10.1.3. 转换函数

trunc(x)	截取实数值 x 的整数部分.
round(x)	舍入实数变元 x 到最接近的整数.
ord(x)	x 必须属于纯量类型 (包括布尔类型和字符类型)[1], 结果(整数类型)是值 x 在 x 的类型所定义的集合中的序号.
chr(x)	x 必须属于整数类型,结果 (字符类型) 是序号为 x 的字符(若它存在的话).

1) 除实数类型外. ——译者注

10.1.4. 其他标准函数

succ (x)	x 属于任何纯量类型[1]或子域类型，结果是 x 的后继值(若有的话).
pred (x)	x 属于任何纯量类型[1] 或子域类型，结果是 x 的前启值(若有的话).

11. 输入和输出

清晰易读的输入和输出的基础是行文文件(参见 5.2.4.)，它们作为程序参数(参见 12.) 传递给 PASCAL 程序，在该程序的环境里表示某些输入输出设备，如象终端、卡片阅读机、或行式打印机等.为了使得行文文件的处理方便，除了 **get** 和 **put** 之外再引进四个标准过程： **read, write, readln** 和 **writeln**. 运用这些标准过程的行文文件不必一定表示输入输出设备，也可以是局部文件. 这些新过程使用时，它们参数表的语法是非标准的，尤其允许参数数目可以变化. 其次，参数不一定属于字符类型，也可以属于某些其他类型. 在这种情况下，数据的传送伴随有隐含的数据转换运算. 如果第一个参数是文件变量，那么它就是要读写的文件. 否则，就自动假定标准文件 **input** 和 **output** 分别为读写时的省缺值. 这两个文件预先说明为

```
var input, output: text
```

就行文文件被所谓行标记符号 (参见 5.2.4.) 分成行来说，行文文件是文件类型中的一种特殊情况. 在读行文文件 f 时，如果文件位置进到了行标记符号上，即读过了一行的最后一个字符，那么缓冲器变量 f↑ 的值就变成空白符，并且标准函

1) 除实数类型外. ——译者注

数 eoln(f)(行的结束)得值为真. 再向前移一次文件位置时，f↑被赋给下一行的第一个字符，而 eoln(f) 得值为假（除非下一行由零个字符组成). 行标记符号并不是字符类型的元素，它只能由过程 **writeln** 生成.

11.1. 过 程 read

过程 **read** 有下列规则，其中 f 标记一个行文文件，v1，…，vn 标记类型为字符、整数（或它的子域）或实数的变量.

1) read(v1，…，vn) 等价于 read(input，v1，…，vn)

2) read(f，v1，…，vn) 等价于 read(f，v1)；…；read(f，vn)

3) 若 v 是字符类型的变量，那么 read(f，v) 等价于 v := f↑；get(f)

4) 若 v 是整数类型(或它的子域)或实数类型的变量，那么 read(f，v) 蕴涵着从 f 中读一个按照 PASCAL 的语法构成的一个数的字符序列(参见 3.)，并把这个数赋给 v. 它跳过该数前面的空白符和行标记符号.

过程 read 也可以用来读一个不是行文文件的文件 f. read(f，x) 此时等价于 x := f↑；get(f).

11.2. 过 程 readln

1) readln(v1，…，vn) 等价于 readln(input，v1，…，vn)

2) readln(f，v1，…vn) 等价于 read(f，v1，…，vn)；readln(f)

3) readln(f) 等价于

```
while not eoln(f) do get(f);
```

get (f)

readln 用于读完当前行并跳到下一行的开始.

11.3. 过 程 write

过程 **write** 有下列规则,其中 f 标记一个行文文件, p1, ···, pn 标记所谓的写-参数, e 标记一个表达式[1], m 与 n 标记类型为整数的表达式.

1) write(p1, ···, pn)等价于 write(output, p1, ···, pn)

2) write(f, p1, ···, pn) 等价于 write(f, p1); ···; write (f, pn)

3) 写-参数有下列三种形式:

e : m　　e : m : n　　e

其中 e 表示要"写"在文件 f 上的值, m 与 n 是所谓字段宽度参数. 值 e 可以是一个数、一个字符、一个布尔值或一个串. 如果它的表示不足 m 个字符, 那么添补适当数目的空白符使得恰好书写为 m 个字符. 假若省略了 m, 那么将假定它为由实现所定义的省缺值. 带有宽度参数 n 的形式仅应用于 e 属于实数类型的场合 (见规则 6)).

4) 如果 e 属于**字符**类型,那么

write (f, e:m)

等价于

f↑ := ' '; put (f);　　(重复 m − 1 次)

f↑ := e; put (f)

注意: m 的省缺值在这情况下为 1.

5) 如果 e 属于**整数**类型(或它的子域),那么将把数 e 的

1) 按下文,它的类型应限为简单类型和紧缩字符数组. ——译者注

十进制表示写在文件 f 上，并在前面添补适当数目的空白符使其符合 m 的规定.

6) 如果 e 属于**实数**类型，那么将把数 e 的十进制表示写在文件 f 上，并在前面添补适当数目的空白符使其符合 m 的规定. 如果省略了参数 n (见规则 3))，那么将选择一个由系数和标度因子组成的浮点表示. 否则得到一个有 n 位小数的定点表示.

7) 如果 e 属于**布尔**类型，那么将把字 TRUE 或 FALSE 写在文件 f 上，并在前面添补适当数目的空白符使其符合于 m 的规定.

8) 如果e是一个(紧缩)字符数组，那么将把串 e 写在文件 f 上，并在前面添补适当数目的空白符使其符合于 m 的规定.

过程 write 也可以用来书写一个不是行文文件的文件 f. write (f, x) 此时等价于 f↑:=x; put(f)[1].

11.4. 过 程 writeln

1) writeln(p1, ···, pn) 等价于 writeln(output, p1, ···, pn)

2) writeln(f, p1, ···, pn) 等价于 write(f, p1, ···, pn); writeln (f)

3) writeln(f) 把行标记符号 (参见 5.2.4.) 添加到文件 f 上.

11.5. 其 他 过 程

page (f)　　在打印行文文件 f 时，它使得跳到新的一

1) 此小段文字，原文在标题 11.3. 之前. ——译者注

页的开头。

12. 程　序

一个 PASCAL 程序除了它的首部外，其他和过程说明有同样的形式.

〈程序〉::=〈程序首部〉〈分程序〉·

〈程序首部〉::= **program**〈标识符〉(〈程序参数表〉);

〈程序参数表〉::=〈标识符〉{,〈标识符〉}

符号 **program** 后面的标识符是程序的名字；它在程序的内部没有进一步的含义. 程序参数标记存在于该程序外面的实体，程序通过它们与其环境通信. 这些实体（通常为文件）被称为是外部的；它们象通常的局部变量一样，必须在构成该程序的分程序中说明.

input 和 **output** 这两个标准文件不可以再说明(见11.)；但在使用它们时，必须在程序首部中作为参数列出. 初始准备语句 reset(input) 和 rewrite(output) 将自动生成，程序员不可以再将其写入程序.

例:

```
program copy(f,g);
var f,g: file of real;
begin reset(f); rewrite(g);
    while not eof(f) do
        begin g↑ := f↑; put(g); get(f)
        end
end.

program copytext (input, output);
var ch: char;
begin
```

```
    while not eof(input) do
    begin
        while not eoln(input) do
            begin read(ch); write(ch)
            end;
        readln; writeln
    end
end.
```

13. 实现和程序互换的标准

研制 PASCAL 的主要动机是需要一种强有力的、灵活的语言，它在大多数计算机上实现都具有相当高的效率．定义它的特色时，并没有参照任何具体的机器，以便于程序的互换．下面推荐的若干限制是打算作为实现者和希望其程序可用于不同的计算机的程序员的准绳．这些标准的目的是要增加下述的可能性：不同的实现将是兼容的，而且程序可以从一台机器转到另一台机器上．

1）标记不同对象的标识符必须在前 8 个字符上可以区分．

2）标号至多由 4 个数字组成．

3）实现者可以对用于定义集合的基类型的大小设定一个界限．（因此，位组合格式的表示方法可以合情合理地用于集合．）

4）打印文件的每一行上，第一个字符可以解释为打印机的控制字符，其含义如下：

空白符：空一行
'0'　：空二行
'1'　：印在下一页的头上
'+'　：不换行(叠印)

以可使用的字符集合来表示 PASCAL 时，应遵循下列规则：

5）字符号(指字的符号)——如象 **begin, end,** 等等——书写为字母序列(不用换码字符括起来)．它们不可以作为标识符使用．

6）空白符、行结束符和注释都看作是分隔符．在任何两个相邻的 PASCAL 符号之间可以出现任意多个分隔符，但有下列限制：在标识符、数和字符号的内部不能有分隔符．

7）在任何一对相邻的标识符、数或字符号之间至少要有一个分隔符．

索　　引

三　画

下标变量 (indexed variable) 6.2.1.
下标类型 (index type) 5.2.1.
子域类型 (subrange type) 5.1.3.

四　画

元素 (element) 7.
元素表 (element list) 7.
引用变量 (referenced variable) 6.3.
开域语句 (with statement) 8.2.4.
文件变量 (file variable) 6.2.3.
文件缓冲器 (file buffer) 6.2.3.
文件类型 (file type) 5.2.4.
无标号语句 (unlabelled statement) 8.
无符号常量 (unsigned constant) 7.
无符号实数 (unsigned real) 3.
无符号数 (unsigned number) 3.
无符号整数 (unsigned integer) 3.

分程序 (block) 9.
分情形标号 (case label) 5.2.2.
分情形标号表 (case label list) 8.2.2.2.5.2.2.
分情形表元素 (case list element) 8.2.2.2.
分情形语句 (case statement) 8.2.2.2.

五　画

正负号 (sign) 3.
记录段 (record section) 5.2.2.
记录变量 (record variable) 6.2.2.
记录变量表 (record variable list) 8.2.4.
记录类型 (record type) 5.2.2.
加型运算符 (adding operator) 7.1.3.

六　画

字段标识符 (field identifier) 6.2.2.
字段表 (field list) 5.2.2.
字段命名符 (field designator) 6.2.2.
字母 (letter) 2.
字母或数字 (letter or digit) 3.
过程标识符 (procedure identifier) 8.1.2.
过程或函数说明 (procedure or function declaration) 9.
过程首部 (procedure heading) 9.
过程说明 (procedure declaration) 9.
过程与函数说明部分 (procedure and function declaration part) 9.
过程语句 (procedure statement) 8.1.2.
成分变量 (component variable) 6.2.
成分类型 (component type) 5.2.1.
因子 (factor) 7.
当语句 (while statement) 8.2.3.1.
如果语句 (if statement) 8.2.2.1.
关系运算符 (relational operator) 7.1.4.

七　画

条件语句 (conditional statement) 8.2.2.

形式参数段 (formal parameter section) 9.
初值 (initial value) 8.2.3.3.
串 (string) 3.
纯量类型 (scalar type) 5.1.1.

八 画

空语句 (empty statement) 8.1.
直到语句 (repeat statement) 8.2.3.2.
实在参数 (actual parameter) 8.1.2.
非紧缩构造类型 (unpacked structured type) 5.2.
表达式 (expression) 7.
参数组 (parameter group) 9.
终值 (final value) 8.2.3.3.
转向语句 (goto statement) 8.1.3.
变量 (variable) 6.
变量标识符 (variable identifier) 6.1.
变量说明 (variable declaration) 6.
变量说明部分 (variable declaration part) 9.
变体 (variant) 5.2.2.
变体部分 (variant part) 5.2.2.
函数标识符 (function identifier) 7.2.
函数命名符 (function designator) 7.2.
函数首部 (function heading) 10.
函数说明 (function declaration) 10.
构造类型 (structured type) 5.2.
构造语句 (structured statement) 8.2.
固定部分 (fixed part) 5.2.2.

九 画

类型 (type) 5.
类型标识符 (type identifier) 5.1.
类型定义 (type definition) 5.
类型定义部分 (type definition part) 9.
项 (term) 7.
重复语句 (repetitive statement) 8.2.3.

结果类型 (result type) 10.
标号 (label) 8.
标号说明部分 (label declaration part) 9.
标度因子 (scale factor) 3.
标识符 (identifier) 3.
标志字段 (tag field) 5.2.2.
语句 (statement) 8.
语句部分 (statement part) 9.
指示字变量 (pointer variable) 6.3.
指示字类型 (pointer type) 5.3.
复合语句 (compound statement) 8.2.1.

十 画

乘型运算符 (multiplying operator) 7.1.
特殊符号 (special symbol) 2.

十 一 画

常量 (constant) 4.
常量标识符 (constant identifier) 4.
常量定义 (constant definition) 4.
常量定义部分 (constant definition part) 9.
控制变量 (control variable) 8.2.3.3.
基类型 (base type) 5.2.3.

十 二 画

赋值语句 (assignment statement) 8.1.1.
循环表 (for list) 8.2.3.3.
循环语句 (for statement) 8.2.3.3.
程序参数表 (program parameters) 12.
程序首部 (program heading) 12.
程序 (program) 12.
集合 (set) 7.
集合类型 (set type) 5.2.3.

十 三 画

数字 (digit) 2.
数字序列 (digit sequence) 3.
数组变量 (array variable) 6.2.1.
数组类型 (array type) 5.2.1.
简单表达式 (simple expression) 7.
简单类型 (simple type) 5.1.
简单语句 (simple statement) 8.1.

十 六 画

整体变量 (entire variable) 6.1.

第二部分　用 户 手 册

K. 詹森　　N. 沃思

0. 引　言

下面的叙述大多都假定读者对计算机术语有了最低限度的了解，对程序的结构也有了一点“感受”。本节的目的就是要使读者加深这种直觉认识。

{程序 0.1

假定每年的通货膨胀率分别为百分之七、八与十，求法郎、美元、英磅、马克或荷兰盾在 1，2，…，n 年里的贬值率。}

```
program inflation(output);

const n = 10;
var i : integer; w1, w2, w3: real;
begin i: = 0; w1 := 1. 0; w2 := 1. 0; w3 := 1.0;
    repeat i:=i + 1;
        w1 := w1 * 1.07;
        w2 := w2 * 1.08;
        w3 := w3 * 1.10;
        writeln(i, w1, w2, w3)
    until i = n
end.
```

```
 1  1.0700000000e + 00    1.0800000000e + 00    1.1000000000e + 00
 2  1.1449000000e + 00    1.1664000000e + 00    1.2100000000e + 00
```

```
3  1.225ϕ43ϕϕϕϕϕe + ϕϕ    1.259712ϕϕϕϕϕe + ϕϕ    1.331ϕϕϕϕϕϕϕϕe + ϕϕ
4  1.31ϕ796ϕ1ϕϕϕϕe + ϕϕ   1.36ϕ48896ϕϕϕϕe + ϕϕ   1.4641ϕϕϕϕϕϕϕϕe + ϕϕ
5  1.4ϕ255173ϕ7ϕϕe + ϕϕ   1.469328ϕ768ϕϕe + ϕϕ   1.61ϕ51ϕϕϕϕϕϕϕe + ϕϕ
6  1.5ϕϕ73ϕ351849e + ϕϕ   1.586874322944e + ϕϕ   1.771561ϕϕϕϕϕϕe + ϕϕ
7  1.6ϕ5781476478e + ϕϕ   1.713824268779e + ϕϕ   1.9487171ϕϕϕϕϕe + ϕϕ
8  1.718186179832e + ϕϕ   1.85ϕ93ϕ21ϕ282e + ϕϕ   2.14358881ϕϕϕϕe + ϕϕ
9  1.83845921242ϕe + ϕϕ   1.999ϕϕ46271ϕ4e + ϕϕ   2.357947691ϕϕϕe + ϕϕ
10 1.95715135729ϕe + ϕϕ   2.158924997273e + ϕϕ   2.59374246ϕ1ϕϕe + ϕϕ
```

一个**算法**或计算机程序是由两个基本部分组成的，一个是对要实现的**动作**的描述，另一个是对这些动作所操作的**数据**的描述．动作是由通常所说的**语句**描述的，数据则是由通常所说的**说明**与**定义**描述的．

程序分为**首部**和主体两部分，其主体称为**分程序**．首部给程序起一个名字，并列出它的**参数**．（这些参数是(文件)变量；表示计算的变元与结果．见第 13 章．）文件“output”这个**参数**总是应该有的．分程序由六段组成．这些段是有次序要求的，除最后一段外都可以为空，它们依次是

〈标号说明部分〉
〈常量定义部分〉
〈类型定义部分〉
〈变量说明部分〉
〈过程与函数说明部分〉
〈语句部分〉

第一段列出了该分程序所定义的全部标号；第二段为某些常量定义它们的同义词，即引入一些标识符，以后用来代替那些常量；第三段是类型定义；第四段是变量定义；第五段定义从属的程序部分(即过程和函数)．语句部分则指明要做的动作．

上述的程序轮廓若用**语法图**来表达，则更为精确．从名

字为程序的图开始，一条通过该图的路径定义了一个语法上正确的程序．每个方框按它的名字引用一个图，然后就由这个图来定义该方框的意义． 圆形框里是终极符号（即实

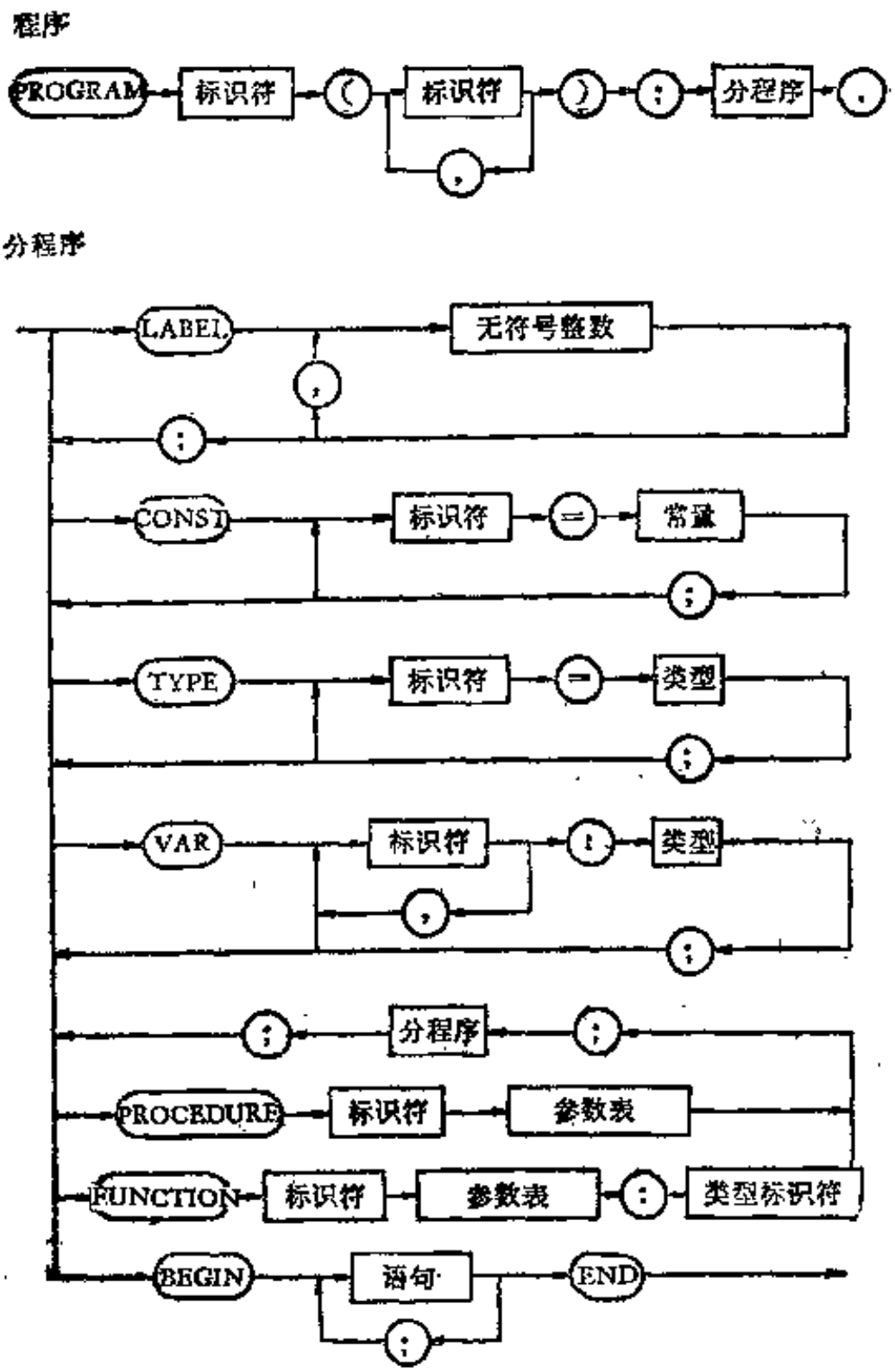


图 0.a　定义一个程序总结构的语法图

际在 PASCAL 程序书写的符号).（完整的 PASCAL 语法图见附录 D.）

语法的另一种表示形式是传统的**巴科斯-瑙尔范式**. 按照这种范式，语法的构成元用括在尖括号〈与〉之间的中文词标记，这些词暗示该成分的性质成含义. 括在元语言括号{与}里构成元的序列，表示它们要重复0次或多次.（PASCAL 的巴科斯-瑙尔范式见附录 D.） 举例来说，图 0.a 中的结构〈程序〉是由下列称为"产生式"的公式定义的.

〈程序〉::＝〈程序首部〉〈分程序〉.

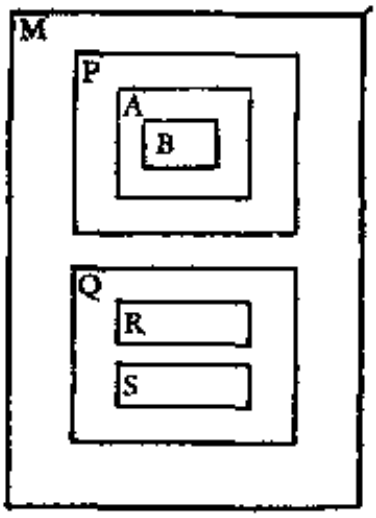
〈程序首部〉:: ＝ **program**〈标识符〉(〈文件标识符〉
{,〈文件标识符〉});

〈文件标识符〉::＝〈标识符〉

每个过程(函数)的结构类似于程序，即它们都由一个首部和一个分程序组成. 所以，过程可以在其他过程里说明(嵌套). 标号、常量同义词、类型、变量以及过程等的说明都**局部**于这些说明所在的过程. 也就是说，它们的标识符仅在构成该过程说明的程序行文里才有意义，并称那段行文为这些标识符的**作用域**. 因为过程可以嵌套，故作用域也可以嵌套. 在主程序里说明的对象，也就是不局部于某一过程的对象，被称为是**全程的**. 它们在整个程序里处处都有意义.

因为分程序可以被过程说明或函数说明嵌套在其他的分程序里，所以对每个分程序都可以给一个嵌套层号. 如果最外面的由程序定义的分程序（即主程序）称为 0 层，则在此分程序里定义的分程序称 1 层；一般说来，在 i 层里定义的分程序将为 i＋1 层. 图 0.b 具体地说明了分程序的结构.

按照前面这种叙述，标识符 x 合法的作用域或范围是 x 的定义所在的分程序的全部，它包括同一分程序中象 x 那样定义的分程序.（在这个例子中要注意所有标识符必须不同.



其中:

0 层 = M

1 层 = P, Q

2 层 = A, R, S

3 层 = B

图 0.b 分程序结构

3.5. 节中将讨论标识符可能相同的情形.

定义对象的分程序	可存取这些对象的分程序
M	M, P, A, B, Q, R, S,
P	P, A, B
A	A, B
B	B
Q	Q, R, S
R	R
S	S

对于熟悉 ALGOL、PL/I 或 FORTRAN 的程序员来说，若从这些语言的角度来粗略地看一下 PASCAL，那可能是有帮助的.为此目的,我们在这里列举了 PASCAL 的一些特征.

1) 变量的说明是强制性的.

2) 一些关键字 (例如 **begin, end, repeat**) 是"保留的",不能作为标识符使用. 它们在本手册中用黑体字表示.

3) 分号(;)看作是语句分隔符,不作为语句终止符(如象 PL/I 那样).

4) 标准数据类型是整数、实数、逻辑值和 (可打印的)

字符这些类型．基本的数据构造工具有数组，记录（对应于COBOL 与 PL/I 的“结构”）、集合和(顺序)文件．这些结构可以组合和嵌套形成集合的数组，记录的文件等结构．数据可以动态地分配，并通过指示字进行存取．应用这些指示字就可以做各种各样的表加工．还可以用符号常量来说明新的基本数据类型．

5）集合数据结构提供了类似于 PL/I 中“位串”的功能．

6）数组可以有任意维数，每一维的界也任意;但数组的界是常量．(即没有动态数组．)

7）如同 FORTRAN、ALGOL 和 PL/I 一样，有转向语句．标号是无符号整数，必须加以说明．

8）复合语句同 ALGOL，它对应于 PL/I 中的循环语句组．

9）ALGOL 的开关和 FORTRAN 的计算转向语句这两个功能由分情形语句体现．

10）循环语句相应于 FORTRAN 的循环语句组，步长只能是 1（**to**）或 －1（**downto**），并仅当控制变量的值落在界内时才执行．因此，它所控制的语句可能根本没有执行．

11）没有条件表达式和多重赋值．

12）过程和函数可以递归调用．

13）变量没有（象 ALGOL 那样的）“固有”属性．

14）参数有值调用和引用调用，没有换名．

15）PASCAL 的“分程序结构”和 ALGOL 与 PL/I 中的分程序结构之差别，是不存在匿名的分程序，即每个分程序都给一个名字并做成一过程．

16）常量、变量等所有对象都必须在它们被引用之前说明．但允许以下两个例外：

a．以指示字类型定义的类型标识符(见第 10 章)；

b. 如果过程调用与函数调用存在向前引用(见第11章).

许多初次接触 PASCAL 的人往往感叹它缺少某些"中意的特色",例如方幂运算符、串的并、动态数组、布尔值上的算术运算、自动的类型转换和省缺说明等. 这都不是疏忽遗漏,却是有意删除的. 在某些情况下,它们总是造成程序设计解法效率不高的主要原因;在另一些情况下,又感到它们总是与程序设计清晰可靠、"风度优美"的目标发生矛盾. 最后,因为可以使用的程序设计功能浩如瀚海,我们必须从中选行严格挑选,以保证编译程序比较紧凑和高效——使得无论是对于仅仅使用语言的一小部分来编写小程序的用户,还是对编写大程序并可能要用到全部语言的用户,都是高效而经济的.

1. 表示法和词汇表

基本词汇表由字母、数字和特殊符号这三类基本符号组成. 特殊符号是运算符和定界符:

```
+    -    *    /    :=    .    ,    ;
:    '    =    <>   <    <=   >=   >
(    )    [    ]    {    }    ↑    ..
```

and	**array**	**begin**	**case**	**const**
div	**do**	**downto**	**else**	**end**
file	**for**	**function**	**goto**	**if**
in	**label**	**mod**	**nil**	**not**
of	**or**	**packod**	**procedure**	**program**
record	**repeat**	**set**	**then**	**to**
type	**until**	**var**	**while**	**with**

字定界符(或称保留字)在手工书写的程序中,通常在其下面打上一条横线,强调它们已被解释为具有固定含义的单

个符号. 程序行文中除 PASCAL 定义明确要求的地方外，程序员不可以使用这些字；特别是，它们不可以用作标识符. 字定界符书写为字母的序列(不用换码字符括起来).

结构

〈〈任意不包含“}”的符号序列〉〉

可以插在任何两个标识符、数或特殊符号之间. 它被称为**注释**，在程序行文中把它们删去后并不改变其含义. 除此之外，符号{与}不再在该语言的任何其他地方使用；在语法描述里，它们象|和::=一样，是标记元语言符号. (在不能使用花括号的系统里，用字符对(*与*)代替.)

标识符是标记常量、类型、变量、过程和函数的名字. 它们必须以字母开始，后面可以有若干任意组合的字母与数字. 虽然标识符可以很长，但在实现时可以给出一个限制：其中多少个字符是有效的. 标准 PASCAL 在实现时将总是认为一个标识符的前八个字符是有效的. 也就是说，标记不同对象的标识符，应当在前八个字符上有所区别.

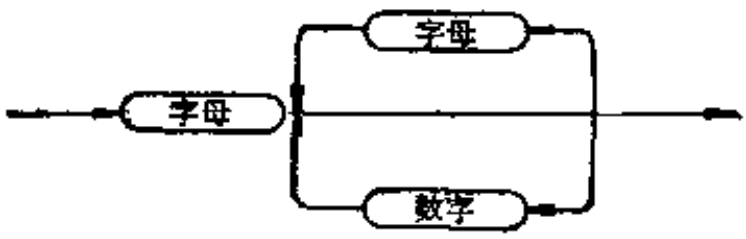


图 1.a 标识符

合法标识符的例子：

sum　　root3　　pi　　h4g　　x

thisisaverylongbutneverthelesslegalidentifier

thisisaverylongbutprobablythesameidentifierasabove

非法标识符的例子：

3rd　　array　　level.4　　root — 3

有些标识符称为**标准标识符**，它们是预先定义的（例如 sin，cos）. 与字定界符（如 **array**）相反，人们不必受这些定义的限制而可以将任何的标准标识符进行重新定义，因为它们被认为是在整个程序的分程序外面的一个假想的分程序里加以说明的.

数采用十进制表示法. 标度因子前的 E 表示"乘以 10 的…幂". 图 1.b 概括了无符号数的语法.

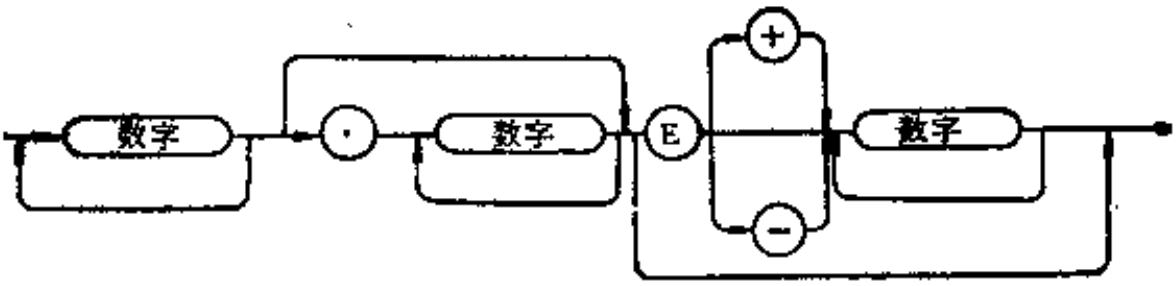


图 1.b 无符号数

注意，如果一个数含有小数点，那么在小数点前后都必须至少有一位数字. 而且，数的内部不可以出现逗号.

无符号数的例子：

3　03　6272844　0.6　5 E — 8　49.22 E + 08　1E 10

写得不正确的数的例子：

3,487,159　XII　.6　E10　5.E — 16

空白符、行结束符与注释都被看作是**分隔符**. 任意两个相邻的 PASCAL 符号之间可以出现任意多个分隔符，但有下列例外：在标识符、数或特殊符号内部不可出现分隔符. 可是，任意一对相邻的标识符、数或字符号之间必须至少有一个分隔符.

用单引号括起来的字符序列称为**串**. 假若一个串内要包含单引号，则把该单引号书写两次.

串的例子：

'a'　';'　'3'　'begin'　'don'　't'

' this string has 33 characters'

2. 数据的概念

数据是描述所有可被计算机操作的通用表达式. 在硬件和机器代码一级上,所有数据均表示为二进制数字(位)的序列. 高级语言允许人们利用数据的抽象概念而不问其如何表示的细节——这主要是由于发展了**数据类型**的概念.

数据类型定义了变量可取的值的集合. 在一个程序中所出现的任何变量都必须与一个,而且仅仅与一个数据类型相联系. 虽然 PASCAL 语言的数据类型可以相当复杂,但是每个数据类型最终一定是由非构造类型所构成的. 非构造类型或者是程序员定义的,并称之为说明的纯量类型,或者是四种标准的纯量类型之一: 整数、实数、布尔或字符.

刻划一个纯量类型的特征是它的值集合,该集合中的元素互不相同而且为它们定义了一个线性次序. 纯量类型的值是由其类型定义中的标识符标记的(见第 5 章).

2.1. 布 尔 类 型

布尔值是一个逻辑真值,它们由预先定义好的标识符false(假)与 true (真)所标记.

下列的逻辑运算符当应用于布尔运算对象时将产生一个逻辑真值: (附录 B 概要说明了所有的运算符).

and	逻辑合取
or	逻辑析取
not	逻辑否定

每一个关系运算符 (=, <>, <=, <, >, >=, **in**)

均产生一布尔值. 其次,布尔类型的定义使得 false<true. 因此,可以利用上面的逻辑运算符和关系运算符定义出 16 种布尔运算中的任何一种. 例如,设 p 与 q 为布尔值,那么

蕴涵	可表达为	p<=q
等价	可表达为	p = q
不可兼或	可表达为	p<>q

标准布尔函数——即产生布尔结果的标准函数——有:(附录 A 概要说明了所有的标准函数.)

odd(x)	当整数 x 是奇数时为真,否则为假.
eoln(f)	一行的结束,在第 9 章中解释.
eof(f)	文件结束,在第 9 章中解释.

2.2. 整 数 类 型

整数类型的值是由实现所定义的整数子集的元素.

下列算术运算符当应用于整数运算对象时产生一个整数值:

*	乘
div	除并截尾(即值不作舍入)
mod	a **mod** b = a − ((a **div** b) * b)
+	加
−	减

关系运算符 =、<>、<、<=、>= 与 > 当应用于整数运算对象时将产生一个布尔结果. < > 标记不等.

四个重要的得到整数结果的标准函数是:

abs(x)	结果是 x 的绝对值
sqr(x)	结果是 x 的平方
trunc(x)	x 是实数值; 结果是它的整数部分.

(舍去小数部分. 因而 trunc (3.7) = 3, trunc (−3.7) = −3.)

round(x) x 是实数值; 结果是经舍入的整数. round(x)表示:在 x>=0 时为 trunc (x + 0.5); 在 x < 0 时为 trunc (x − 0.5).

注意: abs 和 sqr 仅当其变元是整数类型时才产生整数结果.

如果 i 是整数类型的变量,那么

succ(i) 为"下一个"整数

pred(i) 为"前一个"整数

然而,要更清楚的表达,则可以用下列表达式:

i + 1 与 i − 1

PASCAL 有一个依赖于实现的标准标识符 **maxint**[1]. 如果 a 和 b 是整数表达式,那么运算

a **op** b

在

abs(a **op** b) <= maxint

abs(a) <= maxint

abs(b) <= maxint

时保证正确地实现.

2.3. 实 数 类 型

实数类型的值是由实现所定义的实数子集的元素.

只要有一个运算对象属于实数类型 (另一个可能属于整数类型),那么下列运算符将产生一个实数值:

1) 表示一个特定实现所允许的最大整数. ——译者注

*	乘
/	除(运算对象均可以是整数,但结果总是实数)
+	加
−	减

当变元为实数时,下列标准函数产生实数结果.

abs(x)	绝对值
sqr(x)	x 的平方

变元为实数或整数,结果总为实数的标准函数有:

sin(x), cos(x), arctan(x)	三角函数
ln(x)	自然对数
exp(x)	指数函数
sqrt (x)	方根

警告: 虽然实数也称作纯量类型，但它并不总是能用于其他纯量类型可以使用的场合. 特别是，函数 pred 与 succ 不能取实数变元;而且在索引数组、控制循环语句以及定义集合的基类型时,都不能使用实数类型的值.

2.4. 字 符 类 型

字符类型的值是有穷有序字符集的元素. 每一计算机系统为了通信起见,都定义了这样的集合,而且这些字符可以在它的输入输出设备上使用. 遗憾的是，现在还没有一个标准的字符集;因此,字符集中的元素及其次序的确定是与实现紧密相关的.

不论采用何种实现，字符类型都要满足下述最低限度的

假定.

字符集应包括:

1) 按字母顺序排列的大写拉丁字母集 A…Z

2) 按数字顺序排列的并前后相连的十进制数字集 0…9

3) 空白字符.

用撇号(单引号)括起来的一个字符是标记字符类型的常量.

例:

'*'　　'G'　　'3'　　' '　　'X'

(为表示撇号,要把它书写两次.)

两个标准函数 **ord** 与 **chr** 给出了从给定的字符集到自然数子集之间的映射与逆映射——这些自然数称之为字符集的序号; ord 和 chr 称为**转换函数**.

ord(c)　　是字符 c 在基本有序字符集中的序号. (亦见 5.1. 节)

chr(i)　　是序号为 i 的字符值.

我们立即可知, ord 和 chr 互为逆函数,即:

chr(ord(c)) = c,　ord(chr(i)) = i.

此外,一个给定的字符集的次序关系定义为

c1 < c2　　当且仅当　　ord(c1) < ord(c2)

可以把这个定义扩展到每一种关系运算符: =、<>、<、<=、>=、>. 若R标记其中一个运算符,那么

c1 R c2　　当且仅当　　ord(c1) R ord(c2)

当标准函数 pred 与 succ 的变元属于字符类型时, 它们可以定义为

$$pred(c) = chr(ord(c) - 1)$$

$$succ(c) = chr(ord(c) + 1)$$

注意: 一个字符的前启(后继)依赖于基本字符集,它不

存在时则无定义.

3. 程序首部和说明部分

每个程序都由一个首部和一个分程序组成. 分程序包含一个说明部分,它定义所有局部于该程序的对象;分程序还包含一个语句部分,它指明要在这些对象上执行的动作.

〈程序〉::=〈程序首部〉〈分程序〉.

〈分程序〉::=〈标号说明部分〉〈常量定义部分〉
〈类型定义部分〉〈变量说明部分〉
〈过程与函数说明部分〉〈语句部分〉

3.1. 程 序 首 部

首部给程序起一个名字(它在程序内部其他地方并无意义),并列出它的参数. 程序借助于这些参数与其环境进行通信(见 13.2.1. 节).

〈程序首部〉::=**program**〈标识符〉(〈文件标识符〉
{,〈文件标识符〉});

3.2. 标号说明部分

在一个程序里,任何语句的前面均可加上一个标号和冒号(使转向语句可以转向它). 可是,标号在使用前必须在**标号说明部分**中定义. 标号说明部分以符号 **label** 开始,其一般形式为

label 〈标号〉{,〈标号〉};

标号被定义为无符号整数,它至多由四个数字组成.

例:

```
label 3, 18;
```

3.3. 常量定义部分

常量定义引入标识符作为常量的同义词. 符号 **const** 引进常量定义部分,其一般形式为

const〈标识符〉=〈常量〉;
{〈标识符〉=〈常量〉;}

其中常量或是一个数、一个常量标识符(可能带有正负号),或是个串.

使用常量标识符通常使得程序比较好读,并且是文件编制中很方便的辅助工具. 它也使程序员可以把依赖于机器或实例的量集中在程序的开始部分,这样比较醒目,也便于改变. (因而对程序的可移植性和模块化,也有帮助.)

作为一个例子,研究下列程序:

```
{程序 3.1
    常量定义部分的例子}

program convert(output);

const addin = 32; mulby = 1.8; low = 0; high = 39;
      separator =' ----------';
var degree: low..high;
begin
    writeln(separator);
    for degree := low to high do
    begin write(degree, 'c', round(degree * mulby + addin), 'f');
          if odd (degree) then writeln
```

```
    end;
    writeln;
    writeln(separator)
end.
```

```
---------
```

ø c	32 f	1 c	34 f
2 c	36 f	3 c	37 f
4 c	39 f	5 c	41 f
6 c	43 f	7 c	45 f
8 c	46 f	9 c	48 f
1ø c	5ø f	11 c	52 f
12 c	54 f	13 c	55 f
14 c	57 f	15 c	59 f
16 c	61 f	17 c	63 f
18 c	64 f	19 c	66 f
2ø c	68 f	21 c	7ø f
22 c	72 f	23 c	73 f
24 c	75 f	25 c	77 f
26 c	79 f	27 c	81 f
28 c	82 f	29 c	84 f
3ø c	86 f	31 c	88 f
32 c	9ø f	33 c	91 f
34 c	93 f	35 c	95 f
36 c	97 f	37 c	99 f
38 c	1øø f	39 c	1ø2 f

```
---------
```

3.4. 类型定义部分

在 PASCAL 中，数据类型可以直接在变量说明里描述，也可以借助于**类型标识符**引用它们. PASCAL 不仅仅提供了几个标准的类型标识符，而且还提供了**建立新类型**的手段：**类型定义**. 符号 **type** 引进了包含类型定义的程序部分. 类

型定义本身决定一个值集合，并使一个标识符和该集合相联系．它的一般形式是

type〈标识符〉=〈类型〉;{〈标识符〉=〈类型〉;}

类型定义的例子可在以后各章中找到．

3.5. 变量说明部分

在语句中出现的每个变量都必须在**变量说明**里说明．这种说明在行文上必须先于该变量的任何使用．

变量说明通过简单地列出一个标识符并在后面跟随该标识符的类型，把这个标识符和一个数据类型与一个新的变量建立联系．符号 **var** 引出变量说明部分，一般形式是

var〈标识符〉{,〈标识符〉}:〈类型〉;
{〈标识符〉{,〈标识符〉}:〈类型〉;}

例:

```
var root1, root2, root3: real;
    count, i: integer;
    found: Boolean;
    filler: char;
```

这种标识符/类型的联系在包含这个说明的整个分程序里是处处有效的，除非该标识符在从属的分程序里重新被定义．假定分程序 B 嵌套在分程序 A 里．(即 B 在 A 的作用域中说明，因而 B 从属于 A．如图 0.b 所示．）那么可以在 B 中说明一个已在 A 中说明过的标识符，使该标识符与一个局部于 B 的，但在 A 中不能使用的变量相联系；而且这个变量可以属于任何类型．后一个定义在 B 的作用域中处处有效，除非它在从属于 B 的一个分程序里再次说明．不允许一个标识符在同一层或同一作用域里说明多次．因而，下面的说明总是不正确的．

```
var a: integer;
    a: real;
```

3.6. 过程与函数说明部分

每个过程或函数必须在使用前加以说明(或宣布). 过程与函数的说明在 11 章中讨论. 过程是子例行程序,它是由过程语句激活的. 函数是产生结果值的子例行程序,因此它可以作为表达式的组成部分.

4. 动作的概念

动作是计算机程序的根本. 也就是说,一个程序必须用它的数据做点事情——即使那个动作什么事情都不做也罢! **语句**就是描述这些动作. 语句或者是**简单语句**(例如赋值语句),或者是**构造语句**.

4.1. 赋值语句

最基本的语句是**赋值语句**. 它指明把一个新算出的值赋给一个变量. 赋值的形式是

〈变量〉:=〈表达式〉

其中:=是**赋值运算符**,不要把它与关系运算符=相混. 语句"a:=5"读为"a 的当前值替换为值 5",或简单地读为"a **变成** 5".

新值是由计算**表达式**的值得到. 表达式是由常量或变量运算对象、运算符和函数命名符组成的. (函数命名符指明一个函数的激活. 标准函数已列在附录 A 中; 用户定义的函数

将在 11 章中解释.)表达式是计算值的规则,它遵循通常的自左向右和**运算符优先顺序**的规则.运算符 **not**(用于布尔运算对象)具有最高的优先顺序,以下是乘型运算符 (*、/、**div**、**mod**、**and**),然后是加型运算符 (+、-、**or**),而优先顺序最低的是关系运算符(=、<>、<、<=、>=、>、**in**). 任何用括弧括起来的表达式,其计算将不依赖于它前后的运算符.

例:

2*3 — 4*5	= (2*3) — (4*5)	= —14
15 **div** 4*4	= (15 **div** 4)*4	= 12
80/5/3	= (80/5)/3	= 5.333
4/2*3	= (4/2)*3	= 6.0
sqrt (sqr (3) + 11*5)		= 8.0

附录D中的语法反映了优先顺序的实际规则. 读者有疑问时可以随时参考它.

布尔表达式有一种特殊的性质,即它的值在整个表达式的值求完之前就可以知道. 例如,假定有 **x = 0**,那么

(x>0) **and** (x<10)

在第一个因子计算后,已经知道是假,第二个因子没有必要再计算了. 在这种情况下,PASCAL 的规则是既不要求也不禁止第二部分的求值. 这意味着程序必须保证第二个因子是合乎定义的,而且这种合乎定义性不受第一个因子的值的影响. 因此,如果假定数组 a 的下标的界是从 1 到 10,那么下面的例子是错误的!

```
x:=0;
repeat x := x + 1
until (x>10)or (a[x] = 0)
```

(注意: 如果没有一个 a[i] 等于 0,那么该程序将涉及元素 a[11].)

除了文件之外，任何类型的变量都可以赋值．可是，变量（或函数）与表达式必须属于同一类型；但当变量的类型是实数时，表达式的类型可以是整数．（当涉及子域类型时，与它相联系的纯量类型决定赋值的合法性；见 5.2. 节．）

赋值的例子：

```
root1 := pi * x/y
root1 := - root1
root3 := (root1 + root2) * (1.0 + y)
found := y > z
count := count + 1
degree := degree + 10
sqrpr := sqr (pr)
y := sin (x) + cos(y)
```

4.2. 复 合 语 句

复合语句指明它的成分语句要按它们书写的顺序执行．符号 **begin** 与 **end** 起语句括号的作用．注意，一个程序的“主体”有复合语句的形式．

```
{程序 4.1
    复合语句}

program beginend(output);

var sum : integer;
begin
    sum := 3 + 5;
    writeln(sum, -sum)
end.
```

```
        8          -8
```

PASCAL 语言中的分号用于**分隔**语句，它不是语句的终止；即分号不是语句的一部分. 分号的使用规则已清晰地反映在附录 D 的语法里.倘若上例第二个语句的后面添上个分号，那么就认为在这个分号和符号 **end** 之间有一个**空语句**(它的意思是什么动作也没有),这里添个分号没有什么关系，因为这里允许添加空语句. 可是，分号书写位置不对则有可能引起麻烦——参看 4.4. 节的例子.

4.3. 重 复 语 句

重复语句指明某些语句要重复执行. 如果预先（在重复开始之前)知道重复次数,那么循环语句通常是表达这种情形的理想结构;否则应当使用直到语句或当语句.

4.3.1. 当语句

当语句的形式是

while〈表达式〉**do**〈语句〉

这儿的表达式控制重复的次数，必须属于布尔类型. 它在每次重复前都求值;因此,必须使这个表达式尽可能的简单.

```
{程序 4.2
    计算 h(n) = 1 + 1/2 + 1/3 + ··· + 1/n}

program egwhile(input, output);

var n: integer; h: real;
begin read(n); write(n);
    h := 0;
    while n>0 do
        begin h := h + 1/n; n := n - 1
```

```
        end;
    writeln(h)
end.
```

```
        10   2.928968253968e + 00
```

由当语句执行的语句(在上面场合下为复合语句)被重复地执行,直至那表达式的值为假. 如果它的值一开始就为假,那么该语句就根本不执行.

4.3.2. 直到[1] 语句

直到语句的形式是

repeat〈语句〉{;〈语句〉}**until**〈表达式〉

在符号 **repeat** 与 **until** 之间的语句序列至少执行一次. 重复次数由布尔表达式控制,它在每次重复后求值.

{程序 4.3

计算 h(n) = 1 + 1/2 + 1/3 + ··· + 1/n}

```
program egrepeat(input, output);

var n: integer; h: real;
begin read(n); write(n);
    h := 0;
    repeat h := h + 1/n; n := n - 1
    until n = 0;
    writeln(h)
end.
```

```
        10   2.928968253968e + 00
```

1) 原文为"repeat". 为与重复 (Repetive) 语句区别,这里取 "until" 的意思. ——译者注

上面的程序对 n＞0 是正确的. 考虑一下 n≤0 时会怎样? 对这同一个程序改用当语句的形式，那么对所有 n (包括 n = 0) 都是正确的.

注意: 直到语句执行的是语句序列; 若在这里添加括号对 **begin…end**，那将是多余的 (但并不是不正确的).

4.3.3. 循环语句

循环语句表示当一个值级数逐个赋给它的**控制变量**时，重复地执行一个语句. 它的一般形式是

for 〈控制变量〉:=〈初值〉 **to** 〈终值〉 **do** 〈语句〉

(或)

for 〈控制变量〉:=〈初值〉 **down to** 〈终值〉**do** 〈语句〉

{程序 4.4

计算 h(n) = 1 + 1/2 + 1/3 + ··· + 1/n}

```
program egfor (input, output);

var i, n: integer; h: real;
begin read (n); write (n);
    h:=0;
    for i:=n downto 1 do h:=h + 1/i;
    writeln (h)
end.
```

```
        10  2.928968253968e + 00
```

{程序 4.5

利用下述展开式计算余弦:

$$\cos(x) = 1 - x**2/(2*1)$$

$$+x**4/(4*3*2*1)-\cdots\}$$

```
program cosine(input, output);

const   eps = 1e-14;
var     x, sx, s, t: real;
        i, k, n : integer;
begin read (n);
    for i := 1 to n do
    begin read(x); t := 1; k := 0; s := 1; sx := sqr(x);
        while abs(t) > eps*abs(s) do
        begin k := k + 2; t := -t*sx/(k*(k-1)); s := s+t
        end;
        writeln(x, s, k div 2)
    end
end.
```

1.534622222233e-01	9.882477647614e-01	5
3.333333333333e-01	9.449569463147e-01	6
5.000000000000e-01	8.775825618904e-01	7
1.000000000000e+00	5.403023058681e-01	9
3.141592653590e+00	-1.000000000000e+00	14

控制变量、初值与终值必须属于同一纯量类型(除实数类型外),并且这个循环语句不能改变它们. 初值与终值只计算一次. 如果在 **to** (**downto**) 情况下初值大于(小于)终值,那么循环语句不执行. 在循环语句的正常出口时,控制变量的终值无定义.

形式为

for v:=e1 **to** e2 **do** S

的循环语句等价于语句序列

```
if e1 <= e2 then
begin v:=e1; S; v:=succ (v); S;…; v:=e2; S
```

```
        end
        {在这一点上， v 无定义}
```

而形式为

```
            for v:=e1 downto e2 do S
```

的循环语句等价于语句

```
        if e1 >= e2 then
        begin v:=e1;S; v:=pred(v); S;···; v:=e2; S
        end
        {在这一点上， v 无定义}
```

作为最后一个例子，看一看下面的程序。

{程序 4.6

计算 1－1/2＋1/3－…＋1/9999－1/10000. 用四种方法:

1）自左向右逐项地计算;

2）自左向右分别计算所有正项和负项，然后相减;

3）自右向左逐项地计算;

4）自右向左分别计算所有正项和负项，然后相减。}

```
program summing (output);

var s1, s2p, s2n, s3, s4p, s4n, lrp, lrn, rlp, rln: real;
    i: integer;
begin s1 := 0; s2p :=0; s2n := 0; s3 := 0; s4p := 0; s4n := 0;
    for i:=1 to 5000 do
    begin
        lrp: =1/(2*i - 1);          {正项,自左向右}
        lrn: =1/(2*i);              {负项,自左向右}
        rlp: =1/(10001 - 2*i);      {正项,自右向左}
        rln: =1/(10002 - 2*i);      {负项,自右向左}
         s1: =s1 + lrp - lrn;
```

```
        s2p := s2p + lrp; s2n := s2n + lrn;
        s3 := s3 + rlp - rln;
        s4p := s4p + rlp; s4n := s4n + rln
    end;
    writeln(s1,s2p - s2n);
    writeln(s3, s4p - s4n)
    end.
```

```
6.930971830595e - 01    6.930971830612e - 01
6.930971830599e - 01    6.930971830601e - 01
```

为什么四个"完全相等的"和会不同呢?

4.4. 条 件 语 句

条件语句,即如果语句或分情形语句,是从它的成分语句中选择执行其中一个语句. **如果语句**指明一个语句仅当某个条件(布尔表达式)为真时才被执行. 如果为假,则或者什么语句都不执行,或者执行符号 **else** 后面的语句.

4.4.1. 如果语句

如果语句有两种形式

if〈表达式〉**then**〈语句〉

与

if〈表达式〉**then**〈语句〉**else**〈语句〉

在符号 **if** 与 **then** 之间的表达式必须属于布尔类型. 注意,第一种形式可以看成是第二种形式在 **else** 后的语句为空语句时的缩写. 请特别注意: 在符号 **else** 之前决不可以

再有分号！因此，行文

```
if p then begin S1;S2;S3 end; else S4
```

是不正确的．下面的行文恐怕是更靠不住的了：

```
if p then; begin S1; S2; S3 end
```

这里，**if** 所控制的语句是 **then** 与分号之间的空语句；因此，如果语句后面的复合语句将总是被执行．

由结构

```
if〈表达式 1〉then if〈表达式 2〉
    then〈语句 1〉else〈语句 2〉
```

引起的语法二义性的解决办法是，把它解释成等价于

```
if〈表达式 1〉then
    begin if〈表达式 2〉then〈语句 1〉
        else〈语句 2〉
    end
```

读者需要更谨慎的是，草率编写的如果语句可能效率很低．例如有 n 个互相排斥的条件 c1，…，cn，每个都确定一个不同的动作 si．设 P(ci) 是 ci 为真的概率，并假定在 i < j 时有 P(ci)>=P(cj)．那么，效率最高的如果子句序列是

```
if c1 then s1
    else if c2 then s2
        else …
            else if c(n − 1) then s(n − 1)
                else sn
```

当查到某个条件满足并执行它相应的语句时，就完成了整个如果语句，从而跳过了其余条件的检验．

假若“found”是个布尔类型的变量，则

```
if a = b then found := true else found := false
```

是另一种常见的滥用如果语句的情形。其实它可以改写为下面这个极为简单的语句

```
found := a = b
```

{程序 4.7
书写罗马数字}

```
program roman(output);

var x,y: integer;
begin y:=1;
    repeat x:=y; write (x, ' ');
        while x>=1000 do
            begin write('m'); x := x - 1000 end;
        if x>=500 then
            begin write('d'); x := x - 500 end;
        while x>=100 do
            begin write('c'); x := x - 100 end;
        if x>=50 then
            begin write('l'); x := x - 50 end;
        while x>=10 do
            begin write('x'); x := x - 10 end;
        if x>=5 then
            begin write('v'); x := x - 5 end;
        while x>=1 do
            begin write('i'); x := x - 1 end;
        writeln; y := 2*y
    until y>5000
end.
```

```
1 i
2 ii
4 iv
8 viii
```

```
  16 xvi
  32 xxxii
  64 lxiiii
 128 cxxviii
 256 cclvi
 512 dxii
1024 mxxiiii
2048 mmxxxxviii
4096 mmmmlxxxxvi
```

还要注意，如果子句所控制的只是一个语句．因此，需要多个动作时，必须使用复合语句．

下面的程序计算实数 x 的 y 次幂，其中 y 是非负整数．删去该程序的内层当语句，可以得到一个比较简单的，又显然是正确的方案：结果 z 由 x 自乘 y 次获得．注意循环不变体：z*(u**e) = x**y．内层的当语句使 z 与 u**e 不变，它显著地改进了算法的效率．

```
{程序 4.8
    自然指数的幂}

program exponentiation(input, output);

var e, y: integer; u,x,z: real;
begin read(x, y); write(x, y);
    z := 1; u := x; e := y;
    while e>0 do
    begin {z*u**e=x**y, e>0}
        while not odd (e) do
            begin e := e div 2; u := sqr(u)
            end;
        e:=e - 1; z:=u*z
    end;
    writeln(z) {z = x**y}
```

```
end.
```

```
2.000000000000e + 00      7  1.280000000000e + 02
```

下面的程序是画出实值函数 f(x)的曲线，它假定 x 轴垂直走向，要求在与坐标相对应的位置上打印星号．星号的位置按下述方法确定：计算 y = f(x)，乘标度因子 s，把乘积舍入到下个整数，再加上一个常数 h．这个结果值就是星号前面留的空白符数目．

{程序 4.9

函数 f(x) = exp(−x) * sin (2 * pi * x) 的图形表示}

```
program graph1(output);
const d = 0.0625; {1/16, 区间 [x, x + 1] 分为 16 行}
      s = 32; {区间 [y,y + 1] 为 32 个字符宽度}
      h = 34; {x 轴的字符位置}
      c = 6.28318; {2*pi} lim = 32;
var x, y: real; i,n: integer;
begin
    for i := 0 to lim do
        begin x := d*i; y :=exp(-x)* sin(c*x);
        n := round(s*y) + h;
        repeat write(' '); n := n - 1
        until n = 0;
        writeln('*')
    end
end.
```



4.4.2. 分情形语句

分情形语句是由一个表达式(选择器)和一个语句表组成的．每个语句均有一个选择器类型的常量作为它的标号．选择器的类型必须是除实数类型外的纯量类型．分情形语句选择执行其标号等于选择器当前值的那个语句；如果不存在这样的标号，那么其作用没有定义．所选的语句一执行完，控制就转向分情形语句的末尾．其形式是

case〈表达式〉**of**
〈分情形标号表〉:〈语句〉;
…
〈分情形标号表〉:〈语句〉
end

例：（设 **var** i:integer; ch:char;）

```
case i of
    0: x := 0;
    1: x := sin(x);
    2: x := cos(x);
    3: x := exp(x);
    4: x := ln(x)
end

case ch of
    'a', 'b', 'c': ch := succ(ch);
    'd', 'e':      ch := pred(ch);
    'f', 'g':      {无用的情形}
end
```

注意："分情形标号"**不是**通常的标号（见 4.5. 节），它们不能被转向语句引用. 它们的次序是任意的；但是，在一给定的分情形语句内，标号必须互不相同.

虽然分情形语句的效率依赖于实现，但通常是在被选择执行的各语句互相排斥且有类似的选择概率时使用.

4.5. 转 向 语 句

转向语句是一种简单语句，它指示进一步的加工应在程序行文的另一部分里继续，即在那个标号所在的位置上继续.

goto〈标号〉

每个标号（至多四位数字的无符号整数）在程序体内出现之前，必须在标号说明中加以说明. 假若标号 L 是在分程序 A 中说明的，那么它的作用域是分程序 A 的整个行文. 也就是说，在 A 的**语句部分**中**某个**语句可以加前缀 L:. 然后在分程序 A 的**整个**行文里，任何其他语句均可以用转向语句的形

式来引用 L.

例（程序片断）：

```
label 1; {分程序 A}
    ...
  procedure B; {分程序 B}
    label 3;
  begin
  3: writeln('error');
      ...
        goto 3
      ...
        goto 1
  end; {分程序 B}
begin {分程序 A}
  ...
1: writeln('test fails')
  {在分程序A中不允许有 "goto 3"}
end
```

警告：从一构造语句外面转进该语句内的作用**没有**定义．由此，下列各例是不正确的．（编译程序不一定给出出错指示．）

不合法的例子：

1)
```
for i: = 1 to 10 do
    begin S1;
    3: S2
    end;
goto 3
```

2)
```
if p then goto 3;
    ...
if q then 3: S
```

3)
```
procedure P;
```

```
begin  ...
    3: S
end;
begin  ...
    goto 3
end.
```

转向语句应当在不得不打破算法的自然结构这种例外的，或非常的情况下才使用．最好要避免用转向语句来实现语句的规范重复和条件执行，因为这种转移使得程序行文的（静态）结构不能反映计算结构． 其次，行文结构和计算结构（即静态结构和动态结构）之间缺乏对应性，对于程序的清晰性是有害的，并使验证的工作大为困难．如果一个程序员在 PASCAL 程序里使用转向语句，那往往说明他还没有学会用 PASCAL 语言来“思考”（以为还象其他程序设计语言一样，转向语句是一种必不可少的结构）．

5. 纯量类型和子域类型

5.1. 纯 量 类 型

PASCAL 语言的基本数据类型是**纯量类型**．它们的定义是通过枚举标识符的方法来指示一个值的有序集合，这些标识符用于标记值．

type 〈类型标识符〉＝(〈标识符〉{,〈标识符〉});

例：

```
type color = (white, red, blue, yellow, purple, green, orange,
              black);
     sex = (male, female);
     day = (mon, tues, wed, thur, fri, sat, sun);
```

```
operators = (plus, minus, times, divide);
```

不合法的例子：

```
type workday = (mon, tues, wed, thur, fri, sat);
     free = (sat, sun);
```

（因为 sat 的类型是二义的。）

读者已经知道布尔标准类型是如下定义的：

```
type Boolean = (false, true);
```

这个定义自动地暗示了标准标识符 false 与 true 的含义（假和真），并指明 false<true.

关系运算符=、<>、<、<=、>=与>可用于所有的纯量类型，只要运算对象属于同一类型。次序关系由这些常量的列举顺序决定。

以纯量类型为变元的标准函数是

succ(x)	例：	succ(blue) = yellow	x 的后继
pred(x)		pred(blue) = red	x 的前启
ord(x)		ord(blue) = 2	x 的序号

列为第一的那个常量的序号是 0。ord(x) = ord (pred (x))+1.

假设 c 和 c1 属于 color 类型(见前面)，b 属于布尔类型，s1，…，sn 是任意的语句，则下列语句是有意义的。

```
for c := black downto red do s1
while (c1<>c) and b do s1
if c>white then c := pred(c)
case c of
    red, blue, yellow: s1;
    purple: s2;
    green, orange: s3;
    white, black: s4
end
```

5.2. 子 域 类 型

可以定义一个类型为任何其他已经定义的纯量类型的**子域**，并将后一纯量类型称为它的**相联纯量类型**. 子域的定义是直接指出该子域中最小和最大的常量值，但下界必须小于上界. 实数类型不允许有子域类型.

type 〈类型标识符〉=〈常量〉..〈常量〉;

从语义上说，用子域类型来替换它的相联纯量类型，这在任何定义中都是合理的. 而且，所有涉及子域类型值的运算是否有效则完全由它的相联纯量类型决定的. 例如，假定有说明

var a: 1..10;b: 0··30; c: 20··30;

这里 a、b 和 c 的相联纯量类型是整数类型. 因而赋值

a := b; c := b; b := c;

都是合法的语句，虽然它们有时是不能执行的. 因此，整个手册在提到纯量类型时，都假定蕴涵有短语"或它的子域类型"，不再(象报告中那样)处处提及.

例:

```
type days = (mon, tues, wed, thur, fri, sat, sun);
                                          {纯量类型}
  workd = mon..fri;                  {days 的子域类型}
  index = 0··63;                  {整数类型的子域类型}
  letter = 'a'..'2';              {字符类型的子域类型}
```

子域类型为问题的陈述提供了进一步解释的手段. 对实现者来说，子域类型还为节省存贮空间和运行时对赋值进行合法性检查提供了方便. (使用子域类型的例子见程序 6.3.)

6. 构造类型概述，数组类型

在PASCAL语言中，纯量类型和子域类型是非构造类型，其它的类型则都是**构造类型**. 如同构造语句是由其他语句组合而成一样，构造类型也是由其它类型组合而成的. 刻划一个构造类型特征的是其**成分**的类型以及——更为重要的——它的构造方法.

各种构造方法都可以使用一种任选，借以指明数据要采用哪种内部表示方法. 在一个类型定义的前面加上符号 **packed** 是通知编译程序应该节省存贮要求，即使由于必要的紧缩和松展运算而耗费额外的执行时间或可能增加目标代码的长度. 用户的责任是认识到他是否想以效率换取空间. (对效率和存贮空间节省上的实际影响是依赖于实现的，实际上也可能不影响.)

6.1. 数 组 类 型

数组类型由固定数目的(数目在该数组引进时定义)并属于同一类型的成分组成，这后一类型称之为**成分类型或基类型**. 每个成分都可以显式标记，也可以直接进行存取，只要数组变量的名字后面跟以通常所说的**下标**，但下标要用方括号括起来. 下标是可计算的，它们的类型称为**下标类型**. 其次，选择(存取)一个成分所需要的时间并不依赖于造择器(下标)的值；因而，数组被称为是**随机存取结构**.

一个数组的定义既指明它的成分类型，也指明它的下标类型. 一般形式是

type A = **array** [T1] **of** T2;

其中，A是新的类型标识符；T1是下标类型，它是纯量类型或子域类型(但整数类型和实数类型不许作为下标类型)；T2为任意的类型。

例：变量说明　　——和——　赋值

```
memory: array [0..max] of integer      memory[i + j] := x
sick:   array [days] of Boolean        sick[mom] := true
```

(当然，这些例子都假定所用的辅助标识符已有定义。)

{基本程序 6.1
在一给定的表里寻找最大数和最小数}

```
program minmax(input, output);

const n = 20;
var i,u,v,min,max: integer;
    a: array [1..n] of integer;
begin
    {假定该程序在这一点上,数组 a 含有值: 35, 68, 94, 7, 88, -5, -3, 12,
       35, 9, -6, 3, 0, -2, 74, 88, 52, 43, 5, 4}
    min := a[1]; max := min; i := 2;
    while i<n do
    begin u := a[i]; v := a[i + 1];
        if u>v then
        begin if u>max then max := u;
              if v<min then min:=v
        end else
        begin if v>max then max := v;
              if u<min then min := u
        end;
        i := i + 2
    end;
    if i = n then
        if a[n]>max then max := a[n]
```

```
        else if a[n]<min then min := a[n];
    writeln(max, min)
end.
        94          -6
```

{程序 6.2

扩充程序 4.9 使其打印 x 轴}

```
program graph2(output);
const d = ∅.∅625; {1/16,区间 [x,x + 1] 分为 16 行}
        s = 32; {区间 [y,y + 1] 为 32 字符宽}
        h1 = 34; {x 轴的字符位置}
        h2 = 68; {行宽}
        c = 6.28318; {2*pi} lim = 32;
var i,j,k,n: integer; x,y: real;
    a: array [1..h2] of char;
begin for j := 1 to h2 do a[j] := ' ';
    for i := ∅ to lim do
    begin x := d*i; y := exp(-x)*sin(c*x);
        a[h1] := ':';n := round (s*y) + h1; a[n] := '*';
        if n<h1 then k := h1 else k: = n;
        for  j := 1 to k do write(a[j]);
      writeln; a[n] := ' '
    end
end.
```

(考虑如何扩充程序 6.2，使其能打印多个函数——使用数组的与不使用数组的.)

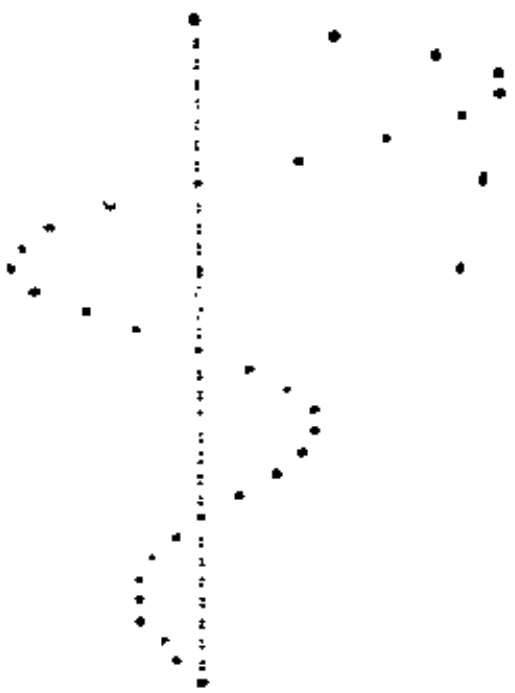
因为 T2 可以是任何类型，所以数组的成分可以是构造类型. 特别，如果 T2 仍是数组类型，那么原来的数组 A 称为**多维数组**. 因此，多维数组M的说明可以写成

var M: **array** [a··b] **of array** [c··d] **of** T

而且

M [i] [j]

标记M的第 i 个成分中的第 j 个成分(类型为 T).



对于多维数组，通常为方便起见可采用缩写

var M: **array** [a..b,c..d] **of** T;

和

M[*i*, j]

我们可以把M看成一个矩阵，并称 M[i, j] 是M的第 i 个成分(第 i 行)的第 j 个(第 j 列的)成分.

并不限于两维，因为T还可以是构造类型. 一般(缩写)形式是

type 〈类型标识符〉= **array** [〈下标类型〉
{ ,〈下标类型〉}] **of**〈成分类型〉;

如果指定了 **n** 个下标类型，那么该数组被称为 **n 维数组**，它的成分由 n 个下标表达式来标记.

{程序 6.3
矩阵乘}

```
program matrixmul(input, output);

const m = 4; p = 3; n = 2;
var i: 1..m; j:1..n; k;1..p;
    s: integer;
    a: array [1..m,1..p] of integer;
    b: array [1..p,1..n] of integer;
    c: array [1..m, 1..n] of integer;
begin {给 a 和 b 赋初值}
    for i := 1 to m do
    begin for k := 1 to p do
          begin read(s); write(s); a[i, k] := s
          end;
          writeln
    end;
    writeln;
    for k := 1 to p do
    begin for j := 1 to n do
          begin read(s); write(s); b[k,j] := s
          end;
          writeln
    end;
    writeln;
    {做 a 乘 b}
    for i := 1 to m do
    begin for j := 1 to n do
          begin s := 0;
                for k := 1 to p do s := s + a[i, k] * b[k, j];
                c[i,j] := s; write(s)
          end;
          writeln
    end;
```

```
    writeln
end.
```

```
 1    2    3
-2    0    2
 1    0    1
-1    2   -3

-1    3
-2    2
 2    1

 1   10
 6   -4
 1    4
-9   -2
```

串在前已定义为由单引号括起来的字符序列（第1章）。一个字符组成的串是标准类型 char（字符）的常量(第2章)；由 n 个字符（n > 1）组成的串被定义为是下列类型的常量：

packed array [1··n] **of** char

在**全同的**数组类型的运算对象之间可赋值（:=）。在全同的紧缩字符数组类型的运算对象上可应用关系运算符 =、<>、<、<=、>与>=，它们的次序关系由基本的字符集决定。

存取紧缩数组的单个成分常常是很浪费的，劝程序员在做紧缩或松展运算时，对整个数组只使用一个运算。这可以借助于标准过程 pack 和 unpack 做到。设A是类型

array [m..n] **of** T

的数组变量，Z是类型

packed array [u..v] **of** T

的变量，其中 (n − m) >= (v − u)，那么

pack(A, i, Z) 意思是 **for** j := u **to** v **do**

Z[j] : = A[j − u + i]

unpack (Z, A, i) 意思是 **for** j : = u **to** v **do**

A[j − u + i] : = Z[j]

其中 j 标记不在程序其他地方出现的辅助变量.

7. 记 录 类 型

记录类型也许是最灵活的数据结构了.从概念上说,一个记录类型是各部分特征可能很不一样的那种结构的模板. 例如,假定我们要记载有关一个人的信息. 已经知道他的姓名、社会安全号、性别、生日、赡养人数与婚姻状况等. 其次,如果那个人是结了婚的或死了爱人的,则要给出(最后一次)结婚的日期; 如果是离了婚的,那要知道(最近一次)离婚的日期,是否第一次离婚;如果是单身的, 那要给出他是否已有独立的住处. 所有这些信息可以用单个"记录"来表达.

较为形式地说, 一个记录是个由固定数目的成分所组成的结构,这些成分称之为**字段**. 它与数组不同,各或分不一定要属于全同的类型, 而且它们也不能直接用下标来检索. 对于每个成分, 记录类型的定义指明了它的类型和一个标记用的标识符——**字段标识符**. 字段标识符的作用域是包含该字段定义的最小记录. 为了使所选成分的类型在程序行文上一目了然(不需要执行程序),故记录选择器由字段标识符组成,它是不变的,不是一个可计算的值.

看一个简单的例子,假定想在 a + bi 形式的复数上进行计算,其 a 和 b 是实数, i 是 −1 的平方根. 我们没有"复数"这种标准类型. 可是, 程序员很容易定义一个记录类型来表示复数. 这个记录大概需要两个字段,都是实数类型,分别表示实部和虚部. 为表达这些所要的语法是

〈记录类型〉::= **record** 〈字段表〉 **end**

〈字段表〉::=〈固定部分〉|〈固定部分〉;
〈变体部分〉|〈变体部分〉

〈固定部分〉::=〈记录段〉{;〈记录段〉}

〈记录段〉::=〈字段标识符〉{,〈字段标识符〉}
:〈类型〉|〈空〉

应用这些规则,可以陈述下列的定义和说明:

```
type complex = record re, im: real end;
var x: complex;
```

其中，complex 是个类型标识符，re 和 im 是字段的标识符，x 是类型 complex 的变量. 因此， x 是由两个成分或字段作成的记录.

类似地,表示日期的变量可以定义为

```
date = record mo: (jan, feb, mar, apr, may, june, july, aug,
                   sept, oct, nov, dec);
              day: 1.. 31;
              year: integer
       end
```

玩具可以定义为

```
toy = record kind: (ball, top, boat, doll, blocks, game, model,
                    book);
            cost: real;
            received: date;
            enjoyed: (alot, some, alittle, none);
            broken, lost: Boolean
      end
```

而课外作业可以定义为

```
assignment = record subject: (history, language, lit, math,
                              psych, science);
             assigned: date;
```

```
        grade: 0..4;
        weight: 1..10
    end
```

如果要引用一个记录成分，则在记录的名字后面跟一个点和相应的字段标识符。例如，下列语句是把 5+3i 赋给 x：

```
x.re := 5;
x.im := 3
```

如果记录本身嵌套在另一个结构内，那么在引用记录成分时用该记录变量的指称作为上述表示中的记录名字。例如，假定希望登记家庭中每个成员最近的天花防疫注射情况。一种可能的办法是把各成员定义为纯量，然后把注射记录定义为记录 date 的数组

```
type family = (father, mother, child1, child2, child3);
var vaccine: array [family] of date;
```

一次更新可以记载为

```
vaccine[child3].mo := apr;
vaccine[child3].day := 23;
vaccine[child3].year := 1973
```

注意：类型“date”也包括(例如)四月三十一日。

{程序 7.1
复数运算}

```
program complex (output);

const fac = 4;
type complex = record re, im: integer end;
var x, y: complex;
    n: integer;
begin
```

```
    x.re := 2; x.im := 7;
    y.re := 6; y.im := 3;
    for n := 1 to 4 do
    begin
        writeln(' x = ', x.re:3, x.im:3,'   y = ',y.re:3, y.im:3);
        {x + y}
        writeln(' sum = ', x.re + y.re:3, x.im + y.im:3);
        {x * y}
        writeln(' product = ', x.re * y.re - x.im * y.im:3,
                               x.re * y.im + x.im * y.re:3);
        writeln;
        x.re:= x. re + fac; x.im:= x. im - fac;
    end
end.
```

```
x  =      2    7      y  =      6    3
sum  =      8    10
product  =     -9    48

x  =      6    3      y  =      6    3
sum  =     12    6
product  =     27    36

x  =     10   -1      y  =      6    3
sum  =     16    2
product  =     63    24

x  =     14   -5      y  =      6    3
sum  =     20   -2
product  =          99    12
```

记录类型的语法还提供了**变体部分**，其意思是可以指明一个记录类型是由若干**变体**组成. 这意味着，不同的变量虽然说属于同一类型，但它们的结构在某些方面可以有所不同. 差别是成分的数目和类型上的不同.

每个变体用一个括在括弧中的表来刻划，表的成分是有

关成分的说明.每个表前面有一个或几个标号;在这些表组成的集合的前面,有一个分情形子句,它指明这些标号的数据类型(也就是说,鉴别各变体用的类型). 例如,假定有一个类型

```
type maritalstatus = (married, widowed, divorced, single)
```

那么可以用下列类型的数据描述一个人 (person):

```
type person = record  〈所有人共同的属性或字段〉
     case marital status of
     married:(〈仅对结了婚的人的字段〉);
     single: (〈仅对单身的人的字段〉);
        ……
     end
```

通常，记录自身有一个成分(字段)是用于表示它当前有效的变体. 例如,上面定义的 person 记录多半是要包含一个公共的字段

ms: marital status

这种常见的情形可以加以缩简，只要在分情形子句中包括起鉴别作用的成分——所谓**标志字段**——的说明，也就是书写为

case ms:maritalstatus **of**

定义变体的语法是

〈变体部分〉::= **case** 〈标志字段〉〈类型标识符〉**of**
〈变体〉{;〈变体〉}

〈变体〉::=〈分情形标号表〉:(〈字段表〉)|〈空〉

〈分情形标号表〉::=〈分情形标号〉{,〈分情形标号〉}

〈分情形标号〉::=〈常量〉

〈标志字段〉::=〈标识符〉:|〈空〉

把有关一个人的信息定义为变体记录结构之前，列出这些信息的"简表"是有益的.

I. 人

A. 姓名(姓,名字)

B. 社会安全号(整数)

C. 性别(男,女)

D. 生日(月,日,年)

E. 赡养人数(整数)

F. 婚姻情况

若结了婚,死了爱人的

a. 结婚日期(月,日,年)

若离婚的

a. 离婚日期(月,日,年)

b. 第一次离婚(假,真)

若单身的

a. 独立居住(假,真)

图 7.a 是两个具有不同属性的"模特儿"人的图.

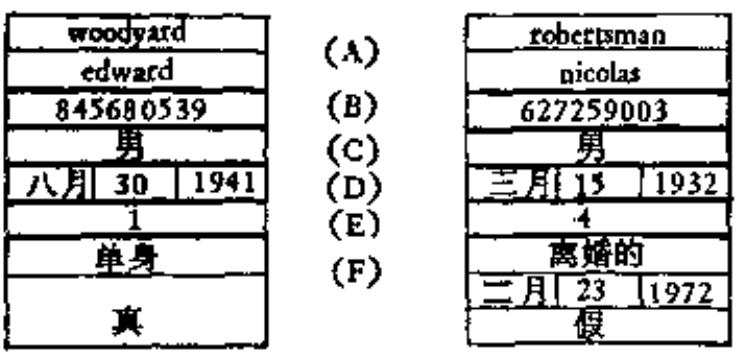


图 7. a 两个模特儿人

定义"人"的记录现在可以系统地叙述为

```
type alfa = packed array [1..10] of char;
     status = (married,widowed, divorced, single);
     date = record mo: (jan, feb, mar, apr, may, jun, july, aug,
                        sept, oct, nov, dec);
            day: 1.. 31;
            year: integer
     end;
```

```
person = record
           name: record first, last: alfa
                 end;
           ss: integer;
           sex: (male, female);
           birth: date;
           depdts: integer;
           case ms:status of
           married, widowed: (mdate: date);
           divorced: (ddate:date; firstd: Boolean);
           single: (indepdt: Boolean)
         end; {person}
```

警告

1）所有字段的名字必须互不相同，即使它们出现在不同的变体中。

2）如果对应于标号 L 的字段表是空的，则形式是

L: ()

3）字段表只能有一个变体部分，并且必须继固定部分之后。（可是，变体部分本身又可以包含变体。因而，变体可能嵌套。）

引用一个记录成分，本质上是简单线性地重新构造其框架。例如，假设变量 P 为类型 person，"建立"上述第一个模特儿人。

```
p.name.last := 'woodyard ';
p.name.first: = 'edward ';
p.ss := 845680539;
p.sex := male;
p.birth.mo := aug;
p.birth.day := 30;
p.birth.year := 1941;
p.depdts := 1;
p.ms := single;
p.indepdt := true
```

7.1. 开 域 语 句

上例的表示方法可能有点冗长，用户可以利用**开域语句**加以简缩．开域子句实际上是打**开**了指定的记录变量的字段标识符所在的作用**域**，使字段标识符可以在变量标识符允许出现的地方出现．（因此，使编译程序有机会优化那个限定语句．）一般形式是

with〈记录变量〉{，〈记录变量〉}**do**〈语句〉

在开域语句的成分语句中，在标记记录变量的字段时只要指定它的字段标识符(它前面不需要整体记录变量的标记)．

下面的开域语句等价于前例那一串赋值语句：

```
with p, name, birth do
begin last := 'woodyard';
    first := 'edward';
    ss := 845680539;
    sex := male;
    mo := aug;
    day := 30;
    year := 1941;
    depdts := 1;
    ms := single;
    indepdt := true
end {with}
```

同样，

```
var currentdate: date;
    ...
with currentdate do
    if mo=dec then
      begin mo := jan; year := year + 1
      end
      else mo := succ(mo)
```

等价于

```
var currentdate: date;
    ...
    if currentdate.mo = dec then
      begin currentdate.mo:=jan;
          currentdate.year:=currentdate.year + 1
      end
    else currentdate.mo:=succ(currentdate.mo)
```

而下面的语句则同样实现了早先例子中的防疫注射记录的修改.

```
with vaccine[child3] do
    begin mo := apr; day: = 23; year := 1973
    end
```

限定语句不能对记录变量表中的任何元素进行赋值. 也就是说,假定有

with *r* **do** S

那么 r 不可以含有任何因 S 而变化的变量;例如:

```
with a[i] do
    begin ...
        i := i + 1
    end
```

是不允许的.

形式为

with r1, r2,···,rn **do** S

的开域语句等价于

```
with r1 do
        with r2 do
    ...
        with rn do S
```

虽则

```
var a: array [2..8] of integer;
    a: 2..8;
```

是不允许的，因为 a 的定义是二义的；但是

```
var a: integer;
    b: record a: real; b: Boolean
       end;
```

却是允许的，因为整数 a 的标记很容易与实数 "b.a" 相区分。同样理由，记录变量 b 也可以与布尔变量 "b.b" 区分开。在

with b **do** *S*

的限定语句 S 中，标识符 a 和 b 现在分别标记成分 b.a 和 b.b。

8. 集 合 类 型

集合类型定义的值集合是它基类型的势集，即基类型值集合的所有子集的集合，包括空集在内。基类型必须是纯量类型或子域类型。

type〈标识符〉 = **set of**〈基类型〉

PASCAL 的实现可以对集合的大小规定界限，它可以很小(例如，一个机器字的位数)。

集合由其元素借助于(语法中由〈集合〉标记的)集合构造器构成。构造的方法是枚举集合中的元素，即枚举属于基类型的表达式；元素间用逗号隔开，最后用集合括号[和]括起来。因此，[]标记空集。

〈集合〉::=[〈元素表〉]

〈元素表〉::=〈元素〉{,〈元素〉}|〈空〉

〈元素〉::=〈表达式〉|〈表达式〉..〈表达式〉

形式 m .. n 是标记基类型中所有满足 m<=i<=n 的元素 i 的集合. 如果 m>n，那么 [m .. n] 标记空集.

集合构造器的例子:

[13]

[i + j,i − j]

['A'..'Z', '0'..'9']

下面的运算符可应用于所有具有集合结构的对象.

+ 并

* 交

− 集合差（例如，A − B 标记 A 中所有不在 B 中的元素的集合.）

可以应用于集合运算对象的关系运算符是

= < > 检查相等

<= >= 检查集合的包含关系



属于(集合)关系. 第二个运算对象属于集合类型，第一个则属于与它相联系的基类型. 当第一个是第二个的元素时，结果为真,否则为假.

例:

说 明	——和—— 赋 值
type primary=(red, yellow, blue); color=set of primary; var huel, hue2 :color;	huel := [red]; hue2 := []; hue2 := hue2 + [succ(red)];
var ch: char; chset1, chset2: set of 'a'…'z';	chset1 := ['d','a','g']; chset2 := ['a'..'z'] − [ch]
var opcode: set of 0..7; add: Boolean;	add := [2, 3] <= opcode

集合运算相当快，且可以用于避免比较复杂的检验。

语句

```
if (ch = 'a') or (ch = 'b') or (ch = 'c')
                or (ch = 'd') or (ch = 'z') then s
```

可以化为比较简单的检验形式：

```
if ch in ['a'..'d', 'z'] then s
```

{程序 8.1

集合运算的例子}

```
program setop(output);

type days = (m, t, w, th, fr, sa, su);
     week = set of days;
var wk, work, free:week;
     d: days;

procedure check(s: week); {过程在11章中介绍}
    var d: days;
begin write(' ');
    for d:=m to su do
        if d in s then write('x') else write('o');
    writeln
end; {check}

begin work:=[ ]; free := [ ];
    wk := [m..su];
    d := sa; free := [d] + free + [su];
    check (free);
    work := wk - free; check(work);
    if free <= wk then write('o');
    if wk >= work then write('k');
    if not(work >= free) then write('jack');
    if [sa] <= work then write(' forget it');
    writeln
end.
```

```
oooooxx
xxxxxoo
ok jack
```

程序的研制

程序设计——在设计算法并系统地描述算法的这一意义上——通常是个复杂的过程，它需要解决大量琐碎的细节问题，还需要掌握一些专门的技术.大概只有在极其特殊的情况下，才只有一种好的解法. 通常，总存在许许多多解决办法，以致于选择最佳程序时需要作全面的分析，不仅要分析各种可用的算法和计算机，而且还要分析该程序在极大多数的情况下将是如何使用的.

因此，构成一个算法将要经过慎重考虑、调查研究和设计决定等一系列步骤. 在早期阶段，注意力最好集中于有关全局的问题，第一次草拟的解法几乎可以不考虑细节问题. 随着设计过程的深化，原来的问题可能分成若干子问题，逐渐比较多地考虑该问题的设计说明书的种种细节以及各种可用工具的特征. 术语**逐步求精**[2]和**结构程序设计**[4]都是和这种方法有关的.

本章以下部分将重新陈述(为了与 PASCAL 的表示方法相一致) C. A. R. Hoare 在**结构程序设计**一书中 [4, "Notes on Data structuring"] 介绍的例子，借此详细地说明一个算法的建立过程.

这个课题是要产生落在范围 2..n 中的素数，其中 n>=2. 在比较了各种算法之后，选中了厄拉多塞的筛选法，因为它比较简单(不用乘除法).

第一个方案是用话描述的:

1) 把所有 2 至 n 中的数放进"筛"中.

2) 选取筛中的最小数，并把它从筛中去掉.

3）把这个数放入“素数表”中.

4）把筛中的数检查一遍，去掉这个数的所有倍数.

5）如果筛不空，则重复 2)—5）步.

虽然设置变量初值是程序执行时的第一步，但在程序的建立过程中它通常是最后的一步. 完完全全地弄清整个算法是正确地设置初值的前提；随着程序一次次改进，为了保证程序正确地运行，修改这些初值是完全必要的.（不幸的是，这种修改并不总是能够满足要求！）

Hoare 选择一个元素为 2..n 的集合类型来表示筛 (sieve) 和素数表 (primes). 下面是他提出的程序方案，我们稍为作了一点更动.

```
const n = 10000;
var sieve, primes: set of 2..n;
    next, j: integer;
begin {设置初值}
    sieve := [2..n]; primes := [ ]; next := 2;
    repeat {找下个素数}
        while not (next in sieve) do next := succ(next);
        primes := primes + [next];
        j := next;
        while j<=n do {消去}
        begin sieve := sieve - [j]; j := j + next
        end
    until sieve = [ ]
end.
```

作为练习，Hoare 要求重写该程序，使得这些集合只表示奇数. 下面的程序是一种建议方案. 注意，它非常接近于第一种解法.

```
const n = 5000; {n' = n div 2}
var sieve, primes: set of 2..n;
    next, j, c: integer;
```

```
begin {设置初值}
    sieve := [2..n]; primes := [ ]; next := 2;
    repeat {找下个素数}
        while not(next in sieve) do next := succ(next);
        primes := primes + [next];
        c := 2 * next - 1; {c = 新素数}
        j := next;
        while j<=n do {消去}
        begin sieve := sieve - [j]; j := j + c
        end
    until sieve = [ ]
end.
```

我们希望所有的基本集合运算都执行得比较快. 许多实现都根据它们的"字长"来限制集合的最大尺寸,使得基本集合的每个元素都可以用一个二进制位表示(0表示不存在,1表示存在). 因此,根大多数的实现都不会允许集合可以有10000个元素. 考虑到这些因素,我们对数据表示进行了调整,如程序 8.2 所示.

大的集合可以表示为较小集合的数组,使得每个小集合都"适应"于一个字(依赖于实现). 下面的程序利用第二个方案作为该算法的抽象模型. 筛 (sieve) 和素数表 (primes) 重新定义为集合的数组;下一个 (next) 定义为记录. 输出部分仍未编制.

{程序 8.2

利用筛选法生成 3 .. 10000 之间的素数,所用的筛只含这个范围内的奇整数}

```
program primes(output);

const wdlength = 59; {依赖于实现}
      maxbit = 58;
```

```
    w = 84; {w = n div wdlength div 2}
var sieve, primes: array [0..w] of set of 0..maxbit;
    next: record word, bit: integer
          end;
    j, k, t, c: integer; empty: boolean;
begin {设置初值}
    for t: = 0 to w do
        begin sieve[t] := [0..maxbit]; primes[t] := [ ] end;
    sieve[0] := sieve[0] - [0]; next.word := 0;
    next.bit := 1; empty := false;
    with next do
    repeat {找下个素数}
        while not (bit in sieve[word]) do bit := succ(bit);
        primes[word] := primes[word] + [bit];
        c := 2 * bit + 1;
        j := bit; k := word;
        while k<=w do{消去}
        begin sieve[k] := sieve [k] - [j];
            k := k + word * 2; j := j + c;
            while j > maxbit do
                begin k := k + 1; j := j - wdlength
                end
        end;
        if sieve[word] = [ ] then
            begin empty := true; bit := 0
            end;
        while empty and (word<w) do
            begin word := word + 1; empty := sieve[word] = [ ]
            end
    until empty; {开域语句结束}
end.
```

9. 文 件 类 型

从多方面看来，最简单的构造方法是序列. 在数据处理的同行中普遍接受的描述序列的术语是**顺序文件**. PASCAL

仅仅使用**文件**这个词，来说明由成分的序列所组成的结构——每个成分都属于同一类型.

各成分的自然次序是由这个序列确定的，并在任何情况下只有一个成分是可以直接存取的. 其他成分则要随文件顺序前移才能存取. 成分的数目称为文件的**长度**，文件类型的定义并没有固定它；这是文件和数组之间一个明显不同的特征. 没有成分的文件称为**空文件**.

type 〈标识符〉 = **file of** 〈类型〉

每个文件变量 f 的说明自动引入一个属于成分类型的**缓冲器变量**，记为 f↑. 可以把它看作一个**窗口**，通过它可以检测(读)现有成分或者添加(写)新的成分，并且若干文件运算符将自动移动这个窗口.

顺序地加工和缓冲器变量的存在，暗示了文件可以与**辅助存贮器**和**外部设备**相联系. 至于各成分实际上如何分配，则是有关实现方面的问题，但我们假定：任何时刻只有若干个成分在主存内，并且只有由 f↑ 指示的成分是可以直接存取的.

当窗口 f↑ 移过了文件 f 的结束符 (eof) 时，标准布尔函数 eof(f) 的值变为真，否则为假. 基本的文件处理运算符有：

reset (f)　文件窗口复原到文件的开始以便读，即 f 第一个元素的值赋给 f↑. 若 f 不空，eof (f) 变成假；否则 f↑ 无定义，eof(f) 仍为真.

rewrite (f)　准备重写文件 f. f 的当前值替换为空文件. eof(f) 变为真，并可以写新文件.

get (f)　文件窗口移到下一成分上；即把这个成分的值赋给缓冲器变量 f↑. 如果不存在下

个成分，那么 eof(f) 变为真，f↑ 的结果值无定义。 get 的作用仅当它执行前 eof(f) 为假时才有定义。

put (f)　把缓冲器变量 f↑ 的值添加到文件 f 上。它的作用仅当它执行前谓词 eof(f) 为真时才有定义。 eof(f) 仍为真，而 f↑ 变成无定义。

原则上，所有生成和检测顺序文件的运算完全可以借助于这四种最基本的文件运算符和谓词 eof 来表达。但实际上，把文件位置移动运算和对缓冲器变量的存取运算结合起来通常是很自然的。我们因此引入两个过程： read 和 write，它们定义如下：

read(f, x)　等价于　x := f↑; get(f)

write(f, x)　等价于　f↑ := x; put(f)

注：报告所定义的标准 PASCAL，提到这些缩写时，仅仅是对 x 属于字符类型的情形。

使用这两个过程的好处不仅在于简短，而且还在于概念上简洁，因为缓冲器变量 f↑ 的值有时没有定义，这个变量是否存在现在可以不管。但是，缓冲器变量作为“先行”设备或许还是有用的。

文件的例子：

说　明	——和—— 语　句
var data: **file of** integer; a: integer;	a := data↑; get (data) read(data, a)
var club: **file of** person; p: person;	club↑ := p; put(club) write(club, p)

程序片断举例：

1) 读一个实数文件f,计算它们的和 S.

```
S := 0; reset (f);
while not eof(f) do
        begin read(f, x); S := S + x
        end
```

2) 下面的程序片断是在两个由整数有序序列

f1, f2,···, fm 和 g1, g2,···, gn

组成的文件 f 和 g 上运算，f 和 g 对任何的 i 和 j 均满足

f(i + 1)>=f(i) 且 g(j + 1)>=g(j).

该程序把这两个文件合并为一个有序文件 h，使

h(k + 1)>=h(k)， 当 k = 1,2,···,(m + n − 1).

该程序使用下列变量：

```
endfg: Boolean;
f, g, h: file of integer
```

```
{基本程序
    合并 f 与 g 为 h}

begin reset(f); reset(g); rewrite(h);
    endfg := eof(f) or eof(g);
    while not endfg do
    begin if f↑<g↑ then
        begin h↑ := f↑; get(f);
            endfg := eof(f)
        end else
        begin h↑ := g↑; get(g);
            endfg := eof(g)
        end;
        put(h)
    end;
    while not eof(g) do
    begin h↑ := g↑; put(h);
```

```
        get(g)
    end;
    while not eof(f) do
    begin h↑:=f↑; put(h);
        get(f)
    end
end
```

文件可以局部于一个程序(或局部于一个过程),也可以已经存在于程序的外面.后者称为**外部文件**.外部文件作为程序首部里的参数(见13章)传递给程序.

9.1. 行 文 文 件

成分为字符的文件称为**行文文件**.与此对应,标准类型text(行文)定义如下:

```
type text = file of char
```

行文通常分成**行**.直截了当地指示两个连续行之间的分隔的方法是利用控制字符.例如,ASCII字符集中利用**cr**(回车)和**lf**(换行)这两个字符来标志一行的结束.可是,许多计算机装置所使用的字符集没有这种控制字符,这意味着必须采用其他方法来表示行的结束.

我们可以认为行文类型是在基类型**char**(只含有可印刷字符)扩充一个(假设的)行分隔符的字符集上定义的.这个控制字符不能赋给字符类型的变量,但是可以由下述特殊的行文文件运算符辨认和生成.

writeln (x)	终止行文文件x的当前行
readln (x)	跳到行文文件x的下一行开始(x↑变成下一行的第一个字符)
eoln (x)	一个布尔函数,它指示行文文件x的当前

行是否到达终点.（如果值为真，那么 x↑对应于行分隔符的位置，但 x↑ 是空白符.）

如果 f 是行文文件，ch 是字符类型变量，那么下列简化的表示方法可以用来代替通用的文件运算符：

缩写形式	展开形式
write(f, ch)	f↑ := ch; put(f)
read(f, ch)	ch := f↑; get(f)

下面的程序图式用上述约定来说明行文文件上的若干典型运算.

1）写一个行文 y. 设 P(c) 计算（下）一个字符并把该字符赋给参数 c. 如果当前行满了，则布尔变量 p 的值为真；如果该行文满了，则 q 的值为真.

```
rewrite(y);
repeat
    repeat p(c); write(y,c)
    until p;
    writeln(y)
until q
```

2）读一个行文 x　设 Q(c) 是标记(下)一个字符 c 的加工，R 是标记碰到行结束符时要执行的动作.

```
reset (x);
while not eof(x) do
begin
    while not eoln(x) do
        begin read(x,c); Q(c)
        end;
    R; readln(x)
end
```

3）把行文 x 复制到行文 y 上，并保持 x 的行结构.

```
reset(x); rewrite(y);
```

```
while not eof(x) do
begin {复制一行}
    while not eoln(x) do
        begin read(x,c); write(y,c)
        end;
    readln(x); writeln(y)
end
```

9.2. 标准文件“input”和“output”

行文文件“input”和“output”通常代表计算机装置的标准 I/O 介质(如卡片阅读机和行式打印机). 因此,它们是计算机和使用计算机的人们之间的主要通信途径.

因为这两个行文文件使用得很频繁,故当行文文件运算里没有明确指示行文文件 f 时,就认为它们是“省缺值”. 也就是说,

等	价	于
write(ch)		write(output,ch)
read(ch)		read(input,ch)
writeln		writeln(output)
readln		readln(input)
eof		eof(input)
eoln		eoln(input)

注意: 文件 input (output) 不能使用标准过程 **reset (rewrite)**.

因此,当 x 是“input”且 y 是“output”的情况下,上述程序图式中前两个可以用下列形式表达:(设 **var** ch: char)

在文件“output”上写一个行文:

```
    repeat
        repeat P(ch); write(ch)
        until p;
        writeln
    until q
```

从文件 "input" 读一个行文:

```
    while not eof do
    begin {加工一行}
        while not eoln do
            begin read(ch); Q(ch)
            end;
        R; readln
    end
```

过程 write 和 read 功能的进一步扩充(为了方便地处理易读的输入输出数据),将在 12 章中描述.

下面两个程序范例说明了行文文件 input 和 output 的使用方法.(请考虑: 如果只使用 get 和 put 即不使用 read 和 write 时,必须作哪些变更.)

{程序 9.1——一个输入文件中字母的频率计算}

```
program fcount (input, output);

var ch: char;
    count: array ['a'..'z'] of integer;
    letter: set of 'a'..'z';
begin letter := ['a'..'z'];
    for ch := 'a' to 'z' do count[ch]: = 0;
    while not eof do
    begin
        while not eoln do
        begin read(ch); write(ch);
            if ch in letter then count[ch] := count[ch] + 1
        end;
```

```
        writeln; readln
    end
end.
```

在某些装置里，当把行文文件发送给打印机时，每行的第一个字符用作打印机控制字符；即这第一个字符并不用于印刷，而是被解释为打印机走纸机构的控制功能。一般都采用下述约定：

空白符	：打印前空一行
'0'	：打印前空二行
'1'	：打印前跳到下页顶上
'+'	：不走纸(即叠印)

下面的程序把空白符插在每行的开始，结果就是通常的一行一行地印刷。

{程序 9.2——插入前端空白符}

```
program insert(input,output);

var ch: char;
begin
    while not eof do
    begin write(' ');
        while not eoln do
            begin read(ch); write(ch)
            end;
        writeln; readln
    end
end.
```

如果使用 read 和 write 时没有指明文件参数，那么省缺约定就假定它们的文件是 input 和 output；此时，文件 input 和 output **必须**在程序首部的参数里提及。

10. 指示字类型

静态变量（静态分配的）是在程序里说明的变量，它们在说明之后就用其标识符来标记. 它们之所以被称为静态的是因为它们在其说明所在的分程序整个执行期间都存在（也就是为它们分配了存贮）. 从另一方面来说，变量也可以借助于过程 **new** 动态地生成（与程序的静态结构没有任何依赖关系）. 这样一种变量从而被称为是**动态变量**.

动态变量并不在显式变量说明中出现，也不能直接用标识符引用它们. 相反，生成一个动态变量是引进一个**指示字**值（它只不过是这个新分配的变量的存贮地址）. 因此，指示字类型 P 是由值的无界集合组成的，这些值均指向给定类型 T 的元素. 我们称 P **受囿于** T. 值 **nil** 总是 P 的元素，但它根本不指向任何元素.

type 〈标识符〉= ↑〈类型标识符〉;

例如，如果 P 是个指示字变量，它由于说明

var p: ↑T

而受囿于类型 T，那么 P 是对类型为 T 的某个变量的引用，并用 p↑ 标记该变量. 标准过程 **new** 就是为了建立或分配这种变量而设置的. 过程调用 new(p) 分配一个类型为 T 的变量，并把它的地址赋给 p.

指示字是一种构造既复杂又灵活的数据结构的简单工具. 如果类型 T 是个记录类型，它含有一个或几个类型为 ↑T 的字段，那么用这种办法就可以构成和任何有穷图等价的结构. 这里，类型为 T 的变量表示结点，指示字是边.

作为一个例子，我们研究关于特定的一群人的"数据库"结构. 假定"人(person)"是由第 7 章中定义的记录表示的，

若象下面那样添加一个指示字类型的字段，就可以形成这种记录的链或链表。

```
type link = ↑person;
     ...
     person = record
                ...
                next: link;
                ...
              end;
```

n 个人的链表可以用图 10.a 描绘。

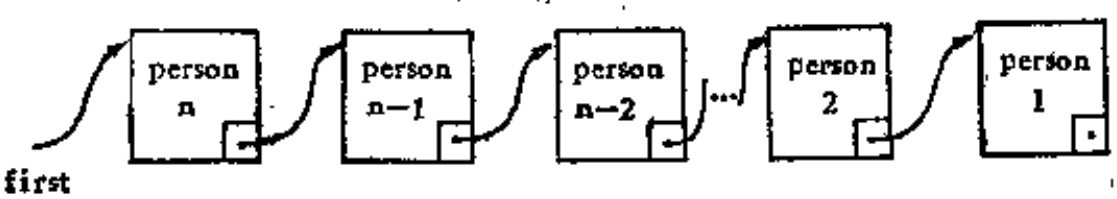


图10.a 链表

一个 link 类型的变量，称为 “first”，(作为链头)指向表的第一个元素。最后一个 person 的链 (next) 是 **nil**.

如果我们假定文件 “input” 含有 n 个社会安全号，那么可以用下述代码来构成上述那样的链.

```
var first, p: link; i: integer;
      ...
    first := nil;
    for i := 1 to n do
    begin read(s); new(p);
      p↑.next := first;
      p↑.ss := s;
      first := p
    end
```

为了便于存取，我们再引进一个类型为 link 的变量，譬如 pt，它可以沿着这个表随便移动. 为了说明如何从表中选择一个记录，假定表中有一个社会安全号等于 n 的 person，

我们要访问它。所用的技巧是将 pt 沿着链移动直至找到所希望的数为止：

```
pt := first;
while pt↑.ss <> n do pt := pt↑.next
```

用言语陈述的话，它是说："让 pt 指向第一个元素。当 pt 指示（引用）的成员的社会安全号不等于 n 时，pt 移动到 pt 当前所引用的记录中的链（也是个指示器变量）所指的变量。"顺便指出：

```
first ↑.next ↑.next
```

存取第三个 person。

注意：仅当确信表中至少存在一个社会安全号等于 n 的人的情况下，这个简单的查找语句才能工作。可是这现实吗？因此被迫要进行检验，以防不知道已经到了表尾。首先可以试试下面这个解决办法：

```
pt := first;
while (pt<>nil) and (pt↑. ss<>n) do
    pt: = pt↑. next
```

可是回想一下 4.1. 节。如果 pt = nil，那么终止条件中第二个因子所引用的变量 pt↑ 根本不存在。下面是两种可能的解决办法，它们正确地处理了这一情况：

```
1)  pt := first; b := true;
    while (pt <> nil) and b do
        if pt↑.ss = n then b := false
                      else pt := pt↑.next

2)  pt := first;
    while pt <> nil do
    begin if pt↑.ss = n then goto 13;
        pt := pt↑.next
    end
```

再考虑一个问题，假定我们希望在数据库中添加一个模特儿人．首先必须用标准过程 **new** 在存贮器内分配一个空间，并为它建立一个引用．

new(p)	分配一新变量 v，并把 v 的指示字引用值赋给指示字变量 p．如果 v 的类型是带变体的记录类型，那么 new(p) 将分配足够的存贮以符合所有变体的要求．形式
new(p,t1,···,tn)	能够为标志字段值等于常量 t1,···,tn 的变体分配一个大小适宜的变量[1]．各标志字段的值按它们说明次序连续地列出．末尾的标志字段的值可以任意省略．这里不包含对标志字段赋值的意思．

警告： 如果记录变量 p↑ 是由 new 的第二种形式建立的，那么这个变量在程序执行期间不可以改变它的变体．对这个整体变量不允许赋值，可是对 p↑ 的成分可以赋值[2]．

对于上面据出的那个问题，在编制某种解法时，第一步是要引进一个指示字变量．假设它是 newp．那么语句

new(newp)

将分配一个类型为 person 的新变量．

下一步，必须把由 newp 所引用的新成员插到由 pt 引用的成员后面．参见图 10.b．

插入是个很简单的事情，只要改变一下指示字：

```
newp↑.next := pt↑.next;
pt↑.next := newp
```

1) 这里指有 n 层嵌套变体的记录结构．——译者注

2) 但对标志字段不能赋不同于调用 new 时所用的值．——译者注

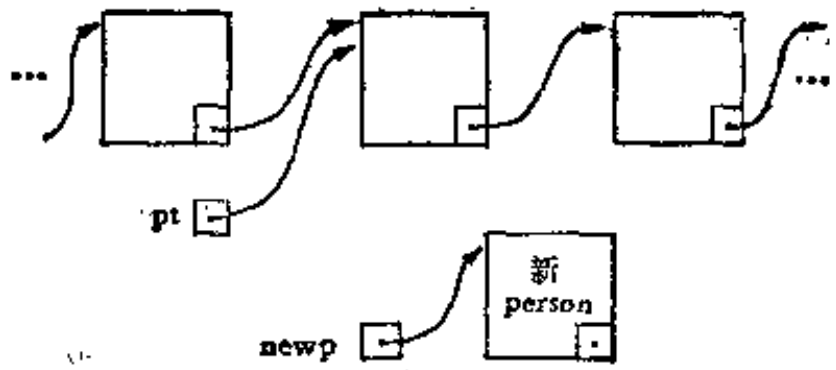


图 10.b 插 入 前

图 10.c 说明了它的结果.

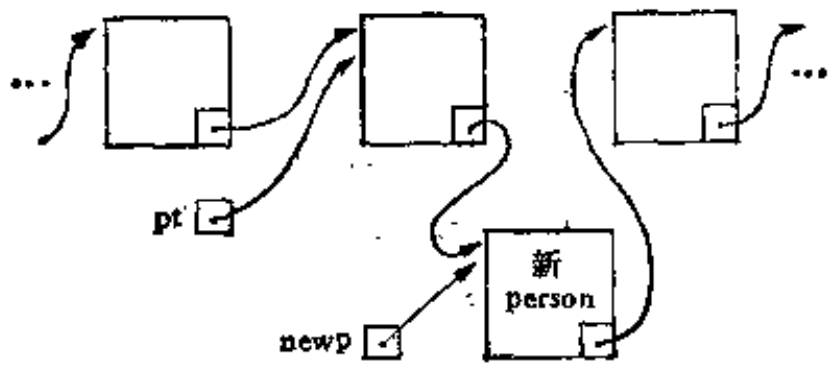


图 10.c 插 入 后

删除辅助指示字 pt 后面的成员只要一条指令:

pt↑.next := pt↑.next↑.next

通常,比较实用的方法是利用两个指示字加工一个表——分别指向两个相邻的成员. 在做删除时多半是一个指示字,例如 p1, 指向要删的成员的前一个, 而 p2 指向要被删除的. 那么删除动作可以用一条指令表达:

p1↑.next := p2↑.next

可是要小心, 在极大多数机器里用这种方法作删除将会丢失可用的(自由)存贮. 一种可能的补救办法是用一个显式表来管理这些"被删除"的元素.这样,新变量将从这个表中(若不空)取,不再调用过程"new";而删除一个表的元素就变

成是把那个元素从该表移到上述这个自由元素表中[1]。

```
p1↑.next := p2↑.next;
p2↑.next := free;
free := p2
```

链式分配是插入和删除元素最有效的表示方法．对数组进行插入（删除）时，则要把某个下标之下的每个元素都向下（向上）移动；而对文件则要求完全重写．

用树结构代替线性结构的例子请参看第11章(程序11.5)．

对聪明的程序员进一言：

PASCAL 提供了极为多种多样的数据结构．留待程序员做的是，对他的问题进行分析，细致到足以决定最适于表达该情况的结构，并且自己对算法进行估价．正如"数据库"的例子所表明的那样，链式分配对于插入和删除特别好．可是如果这些运算不常碰到，而对存取效率有特别的要求，那么用记录的数组来表示数据通常更为恰当些．

11. 过程和函数

随着计算机程序设计技巧的提高，人们在构造程序中都是采用一连串的**求精步骤**．在每一个步骤中，程序员把他的任务分割为若干子任务，从而确定若干个小程序．尽管可以掩饰程序的这种结构，但这并不是人们所希望的．**过程**(或**子例行程序**）这个概念就可以把那些小任务表示为明显的子程序．

1) 在手册的早先版本中，这里提到用 dispose 勾消 new 所分配的变量，并指出：实现者可根据情况用"废码收集"实现或不理睬 dispose 指令。
——译者注

11.1. 过　　程

过程说明用于定义一部分程序，将其与一个标识符相联系，使得这部分程序可通过被一个**过程语句**来激活．过程说明和程序说明有相同的形式，差别是用**过程首部**取代程序首部作为它的开始．

前面讲过一个在整数表中寻找最小数与最大数的基本程序．现在作些扩充，比方说把增量 j1，···，jn 加到 a[1]，···，a[n] 上，而后重新计算最小数和最大数．扩充后的程序运用过程来决定最小数和最大数，程序如下所示。

{程序 11.1
　扩充程序 6.1}

```
program minmax2 (input, output);

const n = 20;
var a: array [1..n] of integer;
    i, j: integer;
procedure minmax;
    var i: 1..n; u,v,min,max: integer;
begin min := a[1]; max := min; i := 2;
    while i<n do
    begin u := a[i]; v := a[i + 1];
        if u > v then
        begin if u > max then max := u;
              if v < min then min := v
        end else
        begin if v > max then max := v;
              if u < min then min := u
        end;
        i := i + 2
```

```
    end;
    if i = n then
        if a[n]>max then max := a[n]
        else if a[n]<min then min := a[n];
    writeln(min, max); writeln
end; {minmax}

begin {读数组}
    for i:=1 to n do
        begin read(a[i]); write(a[i]: 3)
        end;
    writeln;
    minmax;
    for i:=1 to n do
        begin read(j); a[i] := a[i] + j; write(a[i]:3)
        end;
    writeln;
    minmax
end.
```

```
 -1 -3  4  7  8 54 23 -5  3  9  9   9 -6 45 79 79  3  1  1  5
        -6       79

 44 40  7 15  9 88 15 -4  7 43 12 17 -7 48 59 39  9  7  7 12
       -7       88
```

这个程序虽然很简单，但阐明了许多要点：

1）过程首部　最简单的形式，即是

procedure 〈标识符〉;

2）局部变量　局部于过程 minmax 的变量是 i, u, v, min 和 max. 这些变量只能在 minmax 的作用域内引用；若在 minmax 的作用域外面给它们赋值，那么对该程序没有作用.

3）全程变量　全程变量是 a, i 和 j. 程序处处都可以引用它们.（例如，minmax 中第一个赋值是 min:=a[1].）

4）*名字优先顺序*. 注意：i 既是个全程变量的名字，也是个全程变量的名字。它们并不是同一个变量！一个过程可以引用任何对它为全程的变量，也可以把全程变量的名字重新给以定义。 如果一个变量的名字被重新定义了，那么名字和类型之间新的联系关系在做该定义的过程的作用域内有效，而且在这个作用域中与它同名的全程变量不再是可用的（除非作为参数传递进来）. 对局部变量 i 的赋值（例如，i:= i + 2）对全程变量 i 不起作用；并因为 i 是标记局部变量，故全程变量 i 实际上是不能存取的.

好的程序设计习惯是把每个不在某过程之外引用的标识符都严格地说明为局部于该过程的量. 这不仅是好的文件编制方法，而且也提供了额外的安全性. 例如，似乎 i 可以仍为全程变量；可是该程序经过下面的扩充后，在 i 控制的循环里调用了过程 minmax，仍把之定义为全程量会导致不正确的计算.

5）*过程语句*. 在这个例子里，主程序中的语句"minmax"激活了该过程.

稍微仔细地查一下这个例子，就会注意到 minmax 被调用了两次. 由于把这部分程序描述成一个过程——即没有把这部分程序明显地书写两遍——程序员不仅节省了打字的时间，而且也节省了存贮空间. 静态代码只存了一次，而且定义局部变量用的空间只在该过程的执行期间使用.

如果把一个动作陈述为一个过程可以提高可读性，那就不应该犹豫，即使它只调用一次. 把程序编制的各步骤定义为若干过程，可以使得程序更便于通信和验证.

把一个问题分解为若干子例行程序时，必须要做的事情通常是引进些新的变量，用以表示这些子例行程序的变元和结果. 这些变量的效用从程序行文上看，应该是清楚的.

下面的程序扩充了前面的例子，以便使数组的最小值和最大值的计算更加通用一些。

{程序 11.2
扩充程序 11.1}

```
program minmax3(input,output);

const n = 20;
type list = array [1..n] of integer;
var a, b: list;
    i,min1, min2, max1, max2: integer;

procedure minmax(var g:list; var j,k:integer);
    var i: 1..n; u, v: integer;
begin j := g[1]; k := j; i := 2;
    while i<n do
    begin u := g[i]; v := g[i + 1];
        if u>v then
        begin if u>k then k := u;
              if v<j then j := v
        end else
        begin if v>k then k := v;
              if u<j then j := u
        end;
        i := i + 2
    end;
    if i = n then
        if g[n]>k then k := g[n]
        else if g[n]<j then j := g[n];
end; {minmax}

begin {读数组}
    for i := 1 to n do
        begin read(a[i]); write(a[i]:3) end;
    writeln;
```

```
    minmax(a,min1, max1);
    writeln(min1, max1, max1 - min1); writeln;
    for i := 1 to n do
        begin read(b[i]); write(b[i]:3) end;
    writeln;
    minmax(b, min2 max2);
    writeln(min2, max2, max2 - min2);
    writeln(abs(min1 - min2), abs(max1 - max2) ); writeln;
    for i:=1 to n do
        begin a [i] := a[i] + b[i]; write(a[i]:3) end;
    writeln;
    minmax(a, min1, max1);
    writeln(min1, max1, max1 - min1)
end.
```

```
-1 -3  4  7  8 54 23 -5 3  9  9  9 -6 45 79  79  3  1  1  5
       -6       79          85

45 43  3  8  1 34 -8  1  4 34  3  8 -1  3 -2 -4  6  6  6  7.
       -8       45          53
         2      34

44 40  7 15  9 88 15 -4 7 43 12 17 -7 48 77  75  9  7  7  12
       -7       88          95
```

在程序 11.2 中，我们碰到了过程首部的第二种形式

procedure 〈标识符〉(〈形式参数段〉
{;〈形式参数段〉});

形式参数段列举每个形式参数的名字和它的类型. 过程首部的后面是说明部分，它引进局部于该过程的对象.

标号说明部分里的标号以及在形式参数部分、常量定义部分、类型定义部分、变量、过程和函数说明部分中引入的标识符**都局部于**这个过程说明，并称该说明是这些对象的**作用域**. 这些对象在其作用域外面是不知道的. 就局部变量而

论，它们的值在语句部分的一开始是无定义的.

参数提供一种替换手段，使得一个过程被重复执行时在变元上可以有所变化.（例如，对过程 minmax 的调用，二次用于扫描数组 a，一次用于扫描数组 b.）

注意过程首部与过程语句之间的对应关系. 后者包含一个**实在参数的表**，这些实在参数替代过程说明中所定义的并与之对应的**形式参数**. 对应关系由它们在实在参数表和形式参数表中的位置确定. 有四种参数：即值参数、变量参数、过程参数(实在参数是过程标识符）及函数参数（实在参数是函数标识符）.

程序 11.2 说明了**变量参数**的情况. 这种实在参数**必须是一个变量**；对应的形式参数一定前置有符号 **var**，并在过程的整个执行期间都代表这个实在变量. 其次，如果 x1，···，xn 是对应于形式变量参数 v1，···，vn 的实在变量，那么 x1，···，xn 应该是**不同的**变量.

所有的地址计算都在过程调用时做. 因而，如果一个变量是某数组的一个成分，那么它的下标表达式在过程调用时计算.

为了形象地描述存贮分配，对每个变量参数可以从形式参数的名字到对应的实在参数的存贮位置画一个箭头. 任何涉及形式参数的运算于是都直接在实在参数上实现. 每当一个参数是表示过程的结果时（如上例中的 i 和 k），它必须定义为变量参数.

当参数段前面没有符号时，则这一段参数被称为是**值参数**. 此时，实在参数一定是个**表达式**(变量是它的简单情形). 对应的形式参数代表被调用的过程里的局部变量. 这个变量接受对应的实在参数的当前值（也就是表达式在过程调用时的值）作为它的初值. 然后过程可以借助于赋值来改变这个

变量的值;但是决不会影响实在参数的值. 因而,值参数决不能表示计算的结果.

程序 11.3 表明了值参数和变量参数之间的差别.

```
{程序 11.3
    过程参数}

program parameters(output);

var a,b:integer;
procedure h(x: integer; var y: integer);
begin x := x + 1; y := y + 1;
    writeln(x, y)
end;
begin a := 0; b := 0;
    h(a, b);
    writeln(a, b)
end.
```

```
1                    1
0                    1
```

程序 11.2 中，数组 g 里任何一个值都没有变更；也就是说，g 不是结果. 因而似乎把 g 定义为值参数也不会影响最终的结果. 要弄清为什么没有这么做，考察一下实现方法则是有帮助的.

对每个值参数,过程调用都分配一个新的存贮区域;它代表这个局部变量. 实在参数的当前值"复制"到这个地方；过程出口时只是释放这个存贮区域.

如果一个参数不是用于传递过程的结果，通常最好选用值参数. 这种参数引用时比较快，而且还会受到保护以免误改它的数据. 可是对于构造类型的参数（例如数组），则应该

慎重，因为这时复制运算相当耗费时间，而且存放这份复制结果所需要的总存贮量也可能很大。 因为对于程序 11.2 的数组，每个元素只引用一次，所以把该参数定义为变量参数正是我们所希望的。

只要重新定义 n 就可以改变那个数组的维数。如果要使该程序可以用于实数数组，那只需要改变其类型定义和变量说明；那些语句，与数据是否为整数没有关系。

过程标识符在该过程本身的行文中的使用，蕴涵着该过程的**递归**执行。本来就采用递归方法定义的问题，通常都用递归方法解决。下面的问题就是一例。

给定下列符号表达式作为数据：

(a + b) * (c — d)

a + b * c — d

(a + b) * c — d

a + b * (c — d)

a * a * a * a

b + c * (d + c * a * a) * b + a.

它们是根据图 11.a 的语法图形成的，句点表示输入结束。

任务是要构成一个程序，把表达式转换为后缀形式(波兰表示)。其作法是为每个语法结构（表达式、项、因子）都构造一个单独用于转换的过程。因为这些语法结构是递归定义的，所以它们所对应的过程本身也能递归地激活。

{程序 11.4

转换为后缀形式}

```
program postfix(input, output);

var ch :char;
```

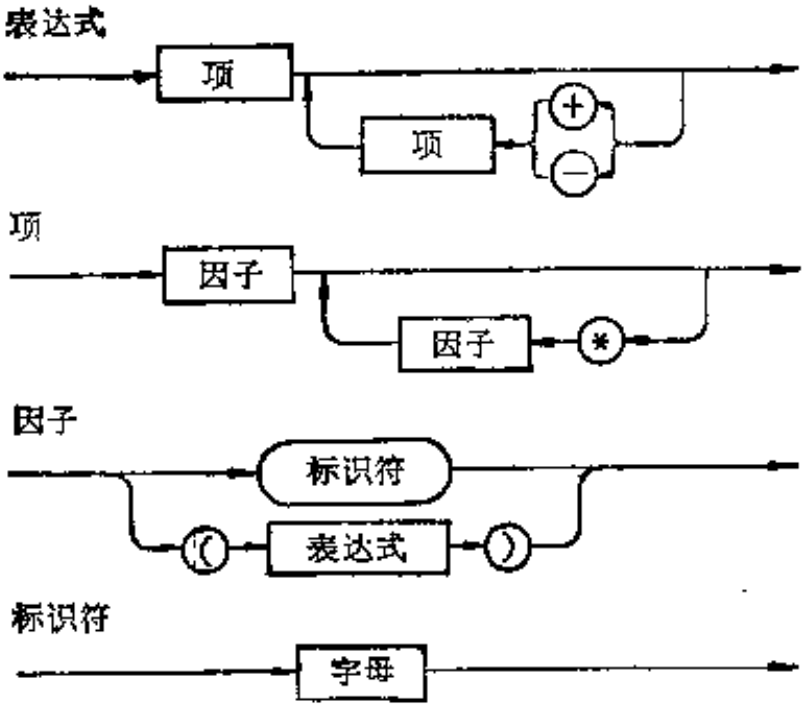


图 11.a 表 达 式

```
procedure find;
begin repeat read (ch)
        until (ch<>'') and not eoln (input)
end;

procedure expression;
    var op: char;

    procedure term;

        procedure factor;
        begin if ch = '(' then
              begin find; expression; {ch = )}
              end else write(ch);
              find
        end; {factor}
    begin factor;
          while ch = '*' do
          begin find; factor; write('*')
          end
    end; {term}
```

```
begin term;
    while (ch = '+') or (ch = '-') do
        begin op := ch; find; term; write(op)
        end
end; {expression}

begin find;
    repeat write(' ');
        expression;
        writeln
    until ch = '.'
end.
```

```
ab+cd-*
abc*+d-
ab+c*d-
abcd-*+
aa*a*a*
bcdca*a*+*b*+a+
```

二叉树这种数据结构本身就是用递归术语定义的，也是用递归算法进行加工处理的. 它是由结点的有穷集合组成；该集合或者为空，或者是由一个带有两枝不相交的二叉树的结点(根)组成. 这两枝二叉树分别称为左子树和右子树[6]. 生成和遍历二叉树用的递归过程自然反映了这种定义方式.

程序11.5建立一个二叉树，并用前缀次序、中缀次序与后缀次序遍历它. 树按前缀次序的形式给出，也就是说列举各结点时(这里为单个字母)，从根开始，接着先左子树再右子树，使得对应于图 11.b 的输入是

abc..ed..fg...hi..jkl..m..n..

这儿句点表示空子树.

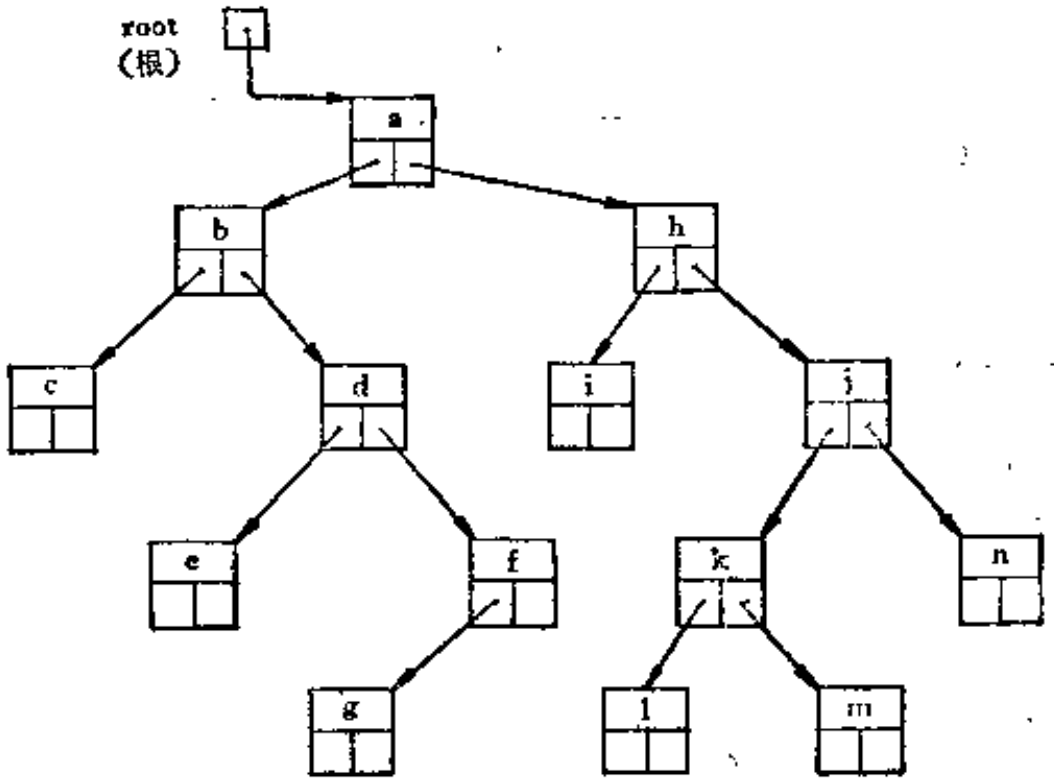


图 11.b　二叉树结构

{程序 11.5
　　遍历二叉树}

```
program traversal(input, output);

type ptr = ↑node;
     node = record info: char;
                   llink, rlink: ptr
            end;
var root: ptr; ch: char;

procedure preorder(p: ptr);
begin if p<>nil then
         begin write(p↑.info);
               preorder(p↑.llink);
               preorder(p↑.rlink)
         end
end; {preorder}
```

```
procedure inorder(p:ptr);
begin if p<>nil then
        begin inorder(p↑.llink);
            write(p↑.info);
            inorder(p↑.rlink)
        end
end; {inorder}

procedure postorder(p: ptr);
begin if p<>nil then
        begin postorder(p↑.llink);
            postorder(p↑.rlink);
            write(p↑.info)
        end
end; {postorder}

procedure enter(var p:ptr);
begin read(ch); write(ch);
    if ch <>'.' then
    begin new (p);
        p↑.info: = ch;
        enter(p↑.llink);
        enter(p↑.rlink);
    end
    else p := nil
end; {enter}

begin
    write(' '); enter(root); writeln;
    write(' '); preorder(root); writeln;
    write(' '); inorder(root); writeln;
    write(' '); postorder(root); writeln
end.

    abc..de..fg...hi..jkl..m..n..
    abcdefghijklmn
```

cbedgfaihlkmjn
cegtdbilmknjha

告诫读者不要不分场合地使用递归技术．虽然看来似乎"精巧"，但它们并不总是能得到最有效的解．

如果过程 P 调用过程 Q，Q 也调用 P，那么不是 P 就是Q必须通过**向前说明**予以"预先宣布"(11.3. 节)．

附录 A 里的**标准过程**，在 PASCAL 的任何实现里都是预先说明了的．任何实现都可添加其他预先说明的过程作为自己的特色．因为它们与所有其他的标准对象一样，都假定是在环绕用户程序的作用域内说明的，所以并不会由于程序中的某个说明对同一标识符进行重新定义而发生冲突．标准过程get、put、read、write、reset 和 rewrite 已在第 9 章中介绍．read 与 write 在第 12 章中还将进一步讨论．

11.2. 函　　数

函数是一部分程序(在与过程同样的意义上)，它计算一个纯量值或指示字值以供求解表达式．**函数命名符**指明一个函数的激活，它由指称该函数的标识**符**和一个实在参数表组成．实在参数是变量、表达式、过程或函数，它们替代对应的形式参数．

函数说明与程序有相同的形式，不同点在**函数首部**，首部的形式是

function 〈标识符〉:〈结果类型〉;

或

function 〈标识符〉(〈形式参数段〉
{;〈形式参数段〉}):〈结果类型〉;

如同过程说明一样，标号说明部分中的标号以及在形式

参数部分、常量定义部分、类型定义部分和变量、过程或函数说明部分里引入的标识符都**局部于**这个函数说明，并称该说明是那些对象的**作用域**. 这些对象在它们的作用域外面是不知道的. 局部变量的值在语句部分开始时未定义。

在函数首部里指定的标识符是为该函数起的名字. 结果类型必须是纯量、子域或指示字类型. 在函数说明内部，必须执行一个赋值，把（结果类型的）值赋给函数标识符. 这个赋值"返回"该函数的结果.

迄今为止，我们的例子只涉及到变量参数和值参数. 同样可以有过程参数和函数参数. 它们必须用特殊的符号引进. 符号 **procedure** 标志形式过程参数；符号 **function** 标志形式函数参数. 下面的程序是用二分法找一个函数的零点，被找的函数在调用时指定.

{程序 11.6
　用二分法找函数的零点}

```
program bisect(input,output);

const eps = 1e - 14;
var x, y: real;

function zero(function f: real; a, b: real): real;
    var x, z: real; s: boolean;
begin s := f(a) < 0;
    repeat x := (a + b)/2.0;
        z := f(x);
        if (z < 0) = s then a := x else b := x
    until abs(a - b) < eps;
    zero := x
end; {zero}
```

```
begin {主程序}
    read(x, y); writeln(x, y, zero(sin, x, y));
    read(x, y); writeln(x,y, zero(cos, x, y))
end.
```

```
-1.000000000000e + 00  1.000000000000e + 00-7.105427357601 e - 15
 1.000000000000e + 00  2.000000000000e + 00  1.570796326795 e + 00
```

对非局部变量或对变量参数的赋值(在函数说明中)称为**副作用**. 这种赋值通常掩饰了该程序的意图，并使验证工作大大复杂化. (于是，有的实现试图禁止副作用.)因此，最好不要使用会产生副作用的函数.

作为一个例子，研究一下程序 11.7.

```
{程序 11.7
    试验副作用}

program sideffect(output);

var a, z : integer;
function sneaky(x: integer): integer;
begin z := z - x; {z 上的副作用}
    sneaky := sqr (x)
end;
begin
    z := 10; a := sneaky(z); writeln(a, z);
    z := 10; a := sneaky(10)*sneaky(z); writeln(a, z);
    z := 10; a := sneaky(z)*sneaky(10); writeln(a, z)
end.
```

```
  100    0
    0    0
10000  -10
```

下面这个例子把程序4.8的取幂算法用函数说明描述。

```
{程序 11.8
    扩充程序 4.8}

program expon2(output);

var pi, spi: real;

function power(x: real; y: integer): real; {y>=0}
    var z: real;
begin z := 1;
    while y>0 do
    begin
        while not odd(y) do
            begin y := y div 2; x := sqr(x)
            end;
        y := y - 1; z := x*z
    end;
    power: = z
end; {power}

begin pi := 3.14159;
    writeln(2.0, 7, power(2.0, 7));
    spi := power(pi, 2);
    writeln(pi, 2, spi);
    writeln(spi, 2, power(spi, 2));
    writeln(pi, 4, power(pi, 4))
end.
```

```
 2.000000000000e + 00    7  1.280000000000e + 02
 3.141590000000e + 00    2  9.869587728100e + 00
 9.869587728100e + 00    2  9.740876192266e + 01
 3.141590000000e + 00    4  9.740876192266e + 01
```

函数标识符在该函数自身内部的一个表达式中的出现，

蕴涵着那个函数的**递归**执行。

{程序 11.9
gcd 的递归描述}

```
program recursivegcd(output);

var x, y, n :integer;

function gcd(m, n: integer): integer;
begin if n = 0 then gcd := m
      else gcd := gcd(n, m mod n)
end; {gcd}

procedure try(a, b: integer);
begin writeln(a, b, gcd(a, b))
end;

begin try(18, 27);
      try(312, 2142);
      try(61, 53);
      try(98, 868)
end.
```

18	27	9
312	2142	6
61	53	1
98	868	14

函数调用可以出现在该函数的说明之前，如果它存在**向前引用**（11.3. 节）的话。

附录 A 中的**标准函数**在 PASCAL 的任何实现中都被假定是预先说明了的。任何实现也可以添加其他预先说明的函数作为自己的特色。

11.3. 附　注

1）过程(函数)调用可以出现在该过程（函数）的定义之前，如果它存在**向前引用**的话。向前引用的形式如下：（注意，参数表和可能有的结果类型**只**在向**前**引用中书写。）

```
procedure Q(x: T); forward;
procedure P(y: T);
    begin
        Q(a)
    end;
procedure Q; {参数表不重复}
    begin
        P(b)
    end;
begin
    P(a);
    Q(b)
end.
```

2）过程与函数在作为其他过程或函数的参数时，只能有值参数。（因而，在运行时没有必要检查一个参数是值调用，还是地址调用。）

3）紧缩结构的成分不可作为实在变量参数。（因此，不需要传递部分字的地址，也不需要在运行时检查实在变量的内部表示。）

4）文件参数必须规定为**变量**参数。

12. 输入和输出

第9章已经提到过人和计算机之间的通信问题。人和计

算机都得学会用某种方法**理解**对方，这种方法称之为**模式识别**．不幸的是，人最容易识别的模式(主要是图形和声音)和计算机能够接受的模式(电脉冲)很不一样．实际上，物理上传输数据的代价——即是指人所认识的模式与计算机所认得的模式之间的转换——可能和处理这些被传输的信息一样昂贵．(因而，许多研究工作都致力于把更多的转换过程"自动化"或"自动进行"，使这种代价减到最小．)这样一种通信任务称之为输入输出处理 (I/O)．

人们可以借助于**输入设备**(例如，键盘穿孔机、卡片阅读机、纸带、磁带、终端等)提交他的信息，并从**输出设备**(例如，行式打印机、卡片穿孔机、纸带穿孔机、终端、直观显示部件等)获得结果，两者的——并由每种装置各自确定的——共同点是易读字符的集合(第 2 章)．正是在这种字符集上，PASCAL 定义了两个标准的行文文件变量(程序参数) **input** 和 **output** (亦见第 9 章)．

行文文件可以用标准文件过程 get 和 put 进行存取．这当然还是很麻烦的，因为这些过程只是为单字符操作而定义的．例如，假定我们有一个自然数，它存于变量 x 中，想把它印在文件 output 上．应当指出，该值在采用十进制表示时和采用罗马数字表示时的字符样式 (见程序 4.7) 很不一样．可是因为人们通常对十进制的表示感兴趣，因而由语言提供一些内部的标准转换过程，把抽象数(不管采用哪种计算机内部表示)翻译为十进制数字的序列，或把十进制数字的序列翻译为抽象数，显然是通情合理的．

为此扩充了 **read** 和 **write** 这两个标准过程，以便简化行文文件的分析与格式编排．调用这两个过程所用的语法是非标准的，因为使用它们时，参数的数目可以变化，参数的类型也不固定．

12.1. 过 程 read

设 v1, v2, ···, vn 表示类型为字符、整数或实数的变量,并假定 f 表示一个行文文件.

1) read(v1,···, vn) 代表

```
read(input, v1, ···, vn)
```

2) read(f, v1, ···, vn) 代表

```
begin read(f, v1); ···; read(f, vn) end
```

3)readln (v1, ···, vn) 代表

```
readln (input, v1, ···, vn)
```

4)readln(f, v1, ···, vn) 代表

```
begin read(f, v1); ···; read(f, vn); readln(f)
end
```

其作用是(从行文文件 f) 读到 vn 后跳过当前行的下余部分.(但 v1, ···, vn 的值可以占据若干行.)

5) 如果 ch 是字符类型的变量,那么 read (f, ch) 代表

```
begin ch: = f↑; get (f) end
```

如果参数 v 属于整数类型(或它的子域)或实数类型,那么读入一串字符,这些字符按 PASCAL 语法是表示一个整数或实数.(相邻的数必须用空白符或行结束符分隔开.)

例:

读入并加工一串数,其中最后一个值后面有一个星号.设 f 是行文文件, x 和 ch 相应为整数(或实数)类型和字符类型的变量.

```
reset(f);
repeat read(f, x, ch);
       P(x)
until ch = '*
```

也许，更为一般的情形是，无法知道要读多少个数据项，也没有用于终止这个数表的特殊符号．这儿有两个可供使用的图式．第一个图式是一项一项地加工．

```
reset(f);
while not eof(f) do
begin read(f, x); skipblanks(f);
    P(x)
end
```

其中 skipblanks(f) 代表下列语句

while (f↑ <>' ') **and not** eof(f) **do** get (f)

第二个图式一次加工 n 个数：

```
reset(f);
while not eof(f) do
begin read(f, x1 ,···, xn); skipblanks(f);
    P(x1, ···, xn)
end
```

(为使这个图式正确工作，总的项数必须是 n 的倍数．)

过程 read 也可用来读不是行文文件的文件(假定 f)．

read(f, x)

此时代表

begin x := f↑; get(f) **end**

12.2. 过 程 write

过程 write 是在行文文件上添加若干字符串（串可以是一个字符或多个字符)．设 p1，···，pn 是参数，其形式在下面定义(见 5)），并设 f 是一行文文件．那么：

1) write(p1，···，pn) 代表

write(output, p1, ···, pn)

2) write(f, p1, ···, pn) 代表

```
begin write(f, p1); ···; write(f, pn) end
```

3) writeln(p1, ···, pn) 代表

```
writeln(output, p1; ···, pn)
```

4) writeln(f, p1, ···, pn) 代表

```
begin write(f, p1); ···; write(f, pn);
    writeln(f)
end
```

其作用是书写 p1, ···, pn, 然后终止行文文件 f 的当前行.

5) 每个参数 pi 必须是下列形式之一:

```
e
e : e1
e : e1 : e2
```

其中 e, e1 和 e2 是表达式.

6) e 是要写的**值**, 它可以属于字符、整数、实数或布尔类型, 也可以是个串. 对于字符类型, write (f, c) 代表

```
f↑ := c; put(f)
```

7) e1 称为最小字段宽度, 它是个任选的控制项. e1 必须是个自然数, 它指示要写入的字符的最小数目. 通常, 值 e 书写为 e1 个字符(不足时在前面补足空白符). 如果是 e1 "太小", 则会多占一些位置. (实数前面至少有一个空白符; 但对整数值没有这个限制.) 如果没有规定字段宽度, 那么根据表达式 e 的类型确定其省缺值(依赖于实现).

8) e2 称为**小数部分长度**, 它也是个任选的控制项, 只应用于 e 属于实数类型的情形. e2 必须是个自然数, 它规定小数点后的数字位数. (于是称这个实数是按定点表示法书写的.) 如果没有规定小数部分的长度, 那么该数以十进制浮点

形式打印.

9) 如果值 e 属于布尔类型,那么就书写标准标识符 true 或 false.

过程 write 也可用于书写不是行文文件的文件(假定f).

write(f, x)

此时代表

begin f↑ := x; put(f) **end**

13. PASCAL 6000-3.4

本章的目的是介绍 CONTROL DATA 6000 计算机上的 PASCAL 所特有的功能. 我们特别提请读者注意, 如果你依靠 PASCAL 6000-3.4 特有的性能,那会使你的程序不能为其他的 PASCAL 实现所接受. 因此劝告大家,只要有可能,尤其在编写"可移植的"程序时, 只运用前面各章所描述的**标准 PASCAL** 的特点.

本章的论题分为四个范畴:

1) 语言的扩充

2) 前面各章尚未确定的规定

3) 限制

4) 附加的预定义过程、函数和类型

13.1. 对 PASCAL 语言的扩充

本节定义 PASCAL 6000-3.4 上可以使用的非标准的语言结构. 虽然它们也许是面向特定操作系统所提供的特殊环境,但它们是用独立于机器的方式描述的,而且按独立于机器的观点也是可以理解的.

13.1.1. 分段文件

一个文件可以看成是分为所谓**段**的，也就是说看成是段的序列；每一个段本身也是个序列。 PASCAL 6000-3.4 提供了一种功能，它可以把一个**文件**说明为**分段文件**，同时能识别各个段和它们的分界。这种文件的每一**段**按照CDC SCOPE术语就是一个"**逻辑记录**"。

说明：

〈文件类型〉:: = **segmented file of** 〈类型〉

例：

```
type T = segmented file of char;
```

谓词函数：

eos (x)　　当文件 x 到达一段的结束位置时值为真，否则为假。

引进下列两个标准过程：

putseg(x)　　在文件 x 生成完一段时必须调用它。

getseg(x)　　它在读文件 x 的下一段时调用。它把下一段的第一个成分赋给缓冲器变量 x↑。 如果没有下一段，那么 eof(x) 变成真；如果有下一段但为空，那么 eos(x) 变成真并且 x↑ 无定义。此后，调用 get(x) 将或者得到下个成分，或者如果不存在下一段使 eos (x) 为真。

当 eos(x) 或 eof(x) 为真时，不可以调用 get(x)；**eof(x)** 总是蕴涵 eos(x)。

分段文件的好处是可以(相当)快速地把读头或写头定到该**文**件的任何段上。为了便于读或(重)写**分**段文件，我们把标准过程 getseg 和 rewrite 加以扩充，使它可以接受两个变元。

getseg(x, n)　准备从文件的当前位置数起的第 n 段上开始读. n > 0 表示向前数; n < 0 表示往回数; n = 0 指示当前段. 注意: getseg (x, 1) 等价于 getseg(x).

rewrite(x, n)　准备从当前位置数起的第 n 段上开始(重)写 x. 注意: rewrite (x,1)不等价于 rewrite (x). 后者是准备从整个文件的头上开始(重)写.

因为文件是为顺序(向前)加工而构造的,所以不应当指望 getseg 与 rewrite 对于 n < 0 的情形会有象 n > 0 时那样的效率.

下面两个带有参数语句 W、R 和 S 的程序图式说明了分段文件的顺序读写运算.

写分段文件 x:

```
rewrite(x);
repeat {生成一段}
    repeat {生成一个成分}
        W(x↑); put(x)
    until p;
    putseg(x)
until q
```

注意: 这个图式不会生成空文件,也不会生成空段.

读分段文件 x:

```
reset(x);
while not eof(x) do
begin {加工一段}
    while not eos(x) do
    begin {加工一个成分}
    R(x↑); get(x)
        end;
```

```
            S; getseg(x)
    end
```

下例展示的过程读分段文件 f，并把每段的前 n 行复制到文件 output 上.

```
procedure list;
    var i, s:integer;
begin s := 0; reset(f);
    while not eof(f) do
    begin s := s + 1; i:=0;
        writeln('segment', s);
        while not eos(f) and (i<n) do
        begin i := i + 1; {复制一行}
            while not eoln(f) do
                begin write(f↑); get(f) {下个字符}
                end
            writeln; readln(f) {下一行}
        end;
        getseg (f) {下一段}
    end
end
```

标准过程 **read** 和 **write** 也可以应用于分段文件.

13.1.2. 外部过程

PASCAL 6000-3.4 提供了一种存取外部过程——即存在于用户程序之外并已单独编译好的过程（函数）——的手段. 这就使 PASCAL 程序员可以访问程序库. 这种过程的说明是由过程首部后面跟一个词 "extern" 或 "Fortran" 组成.

13.2. 前面各章中尚未确定的规定

13.2.1. 程序首部和外部文件

一个 PASCAL 文件变量被实现为 CDC 操作系统里的一个文件. 局部文件分配在磁盘存贮器或扩充的磁心存贮器

(ECS) 上. 它们的存贮在生成时分配，并当它们所局部的那个分程序终止时自动释放.

在程序外面(也就是该程序执行前后)存在的文件也可以在程序中使用，只要把它们指定为控制卡记录的程序调用语句(EXECUTE)中的**实在参数**. 这种文件称为**外部文件**，它们将替代**程序首部**里指明的**形式参数**. 程序首部的形式如下：

program 〈标识符〉(〈程序参数〉
{,〈程序参数〉});

其中程序参数是

〈文件标识符〉或〈文件标识符〉*

这些参数是形式文件标识符；一定要把它们说明为主程序里的文件变量，说明方法与实在局部文件变量完全一样.

由形式参数 **input** 与 **output** 所标记的文件有点特殊，必须注意下列规则：

1) 程序首部**必须**包含形式参数 output.

2) 与所有其它的外部文件不同，这两个形式文件标识符 **input** 与 **output** **不可以**再在说明中定义，因为已自动假定它们的说明是

```
var input, output: text;
```

3) 过程 reset 与 rewrite 当应用于实在文件 INPUT 和 OUTPUT 时不起作用.

例：

```
program P(output, x, y);
...
var x, y: text;
```

如果控制卡记录的 EXECUTE 语句中的实在参数为空，那就假定程序首部里对应的形式参数是实在的“逻辑文件名字”. 例如，当调用一个其首部为

program P(output, f, g);

的程序时，EXECUTE,(,X,) 等价于 EXECUTE, P (OUTPUT, X, G). 建议大家完完全全地规定文件参数，因为依赖省缺值常常会导致本来很容易就可以避免的错误。

13.2.2. 文件的表示

就外部文件而论，重要的是要知道 PASCAL 编译程序所选用的文件表示方法. PASCAL-6000文件的每个成分占据一个 60 位字整数，但成分类型为**字符**的文件（**行文文件**）例外. PASCAL 的行文文件采用 CDC 行文文件约定所规定的“标准”表示: 10 个字符紧缩为一个字. 这意味着在行文文件上使用过程 put 与 get 时要涉及紧缩与松展运算. 用一个右端至少有 12 位 0 的字表示一行的结束. 对于来自**卡片叠**的文件，采用和一般行文文件同样的约定. 注意: 操作系统在读卡片时已经把大部分（但不一定全部）尾随的空白符**删除**了. 因而，组成这种文件的“**卡片映象**”**不**一定有 80 字符.

不分段的文件书写为单个“逻辑记录”(按 SCOPE 术语). 当读一个不分段的外部文件时，记录结束符忽略不计[有一个例外，见下面 3)]. 在分段文件里，每一段对应于一个“逻辑记录”. 没有提供用于指明“记录层”的手段.

外部文件的使用:

1) 如果要读(写)一个外部文件，则对于不分段的文件，读(写)必须用语句

reset(x)　　(rewrite(x))

作初始准备;对于分段文件,则用

reset(x)　　(rewrite(x))

或

getseg(x, n)　　(rewrite(x, n)).

作初始准备.（对于由形式参数 **input** 和 **output** 标记的文件，自动蕴涵这种语句，程序员不必再指明.）

2）每个外部文件是通过调用操作系统例行程序 OPE 而自动“打开”的. 如果这种打开要限制为读的功能——例如对不许写的永久文件——那么必须在程序首部中该文件参数后面标上星号，以表明这种限制. 星号本身并不构成对文件的写保护.

例：

```
program testdata(output, data*);
        ...
var data: file of real;
    r: real;
      ...
    r := data↑ ; get (data)
        ...
```

3）如果用实在文件名字 INPUT 替代某个形式程序参数，譬如 f，那么 f 是文件 INPUT 的当前一个逻辑记录.

13.2.3. 标准类型

13.2.3.1. INTEGER（整数）

标准标识符 **maxint** 定义为

```
const maxint = 281474976710655; { = 2**48 - 1}
```

可是读者要小心，CDC 计算机不提供任何溢出指示. 程序员有责任在可能出现溢出的地方给予检验.

实际上，这台机器能存放的整数可以达到绝对值 2**59，可是对这个范围内的数只有加（+）、减（-）、取绝对值、乘除 2 的方幂（用移位实现）和比较运算才能正确地执行（只要不出现溢出）. 事实上，甚至在 abs(i) > maxint 时都不能印出整数值 i. 但下面这种检验是允许的：

if abs (i) > maxint **then** write ('too big')

13.2.3.2. **REAL**（实数）

实数类型是按照 CDC 6000 的浮点格式定义的. 尾数 48 位，相当于 14 位十进制数字. 最大绝对量级是 10**322.

CDC 的 64 元素的科学字符集

ø	ø	1	2	3	4	5	6	7	8	9
ø	:	A	B	C	D	E	F	G	H	I
10	J	K	L	M	N	O	P	Q	R	S
20	T	U	V	W	X	Y	Z	ø	1	2
30	3	4	5	6	7	8	9	+	−	*
40	/	(	)	$	=		,	.	≡	[
50	]	%	≠	↵	∨	∧	↑	↓	<	>
60	≤	≥	¬	;						

注:

—— ø码在 63 字符集中没有使用
—— 51 码在 63 字符集中为:
—— 48 码在 ETH[1] 为'
—— 53 码在 ETH 为 {
—— 57 码在 ETH 为 }

ASCII 字符集（按 CDC 次序）

	ø	1	2	3	4	5	6	7	8	9
ø	:	A	B	C	D	E	F	G	H	I
10	J	K	L	M	N	O	P	Q	R	S
20	T	U	V	W	X	Y	Z	ø	1	2
30	3	4	5	6	7	8	9	+	−	*
40	/	(	)	$	=		,	.	#	[
50	]	%	"	_	!	&	'	?	<	>
60	@	\	^	;						

图 13.a CDC 字符集

1) ETH 为瑞士工学院缩写. ——译者注

13.2.3.3. CHAR（字符）

字符类型的值是特定装置所提供的字符集中的元素．有下面三种字符集：

1）CDC 科学 64-字符集；

2）CDC 科学 63-字符集；

3）CDC ASCII 64-字符集．

图 13.a 列出了可以使用的字符，并指出了它们的次序；**注意**：CDC 规范所定的 ASCII 字符次序不同于国际标准(ISO)！

基于上面的字符集，PASCAL 6000-3.4 编译程序对某些标准语言符号采用下图所示的表示方法．左面一栏为标准语言符号，右面是它们对应的同义符号．

标准 PASCAL	CDC 科学字符集	ASCII
not	¬	&
and	∧	
or	∨	
<>	≠	
<=	≤	
>=	≥	

图 13.b　标准符号的替代表示

13.2.4. 标准过程 write

如果没有指明最小字段宽度这个参数，则假定为下列省缺值：

类　　型	省　缺　值
integer（整数）	10
real（实数）	22（其中指数总以 E±999 形式表示）
Boolean（布尔）	10
char（字符）	1
string（串）	该串的长度
alfa（α，见 13.4.1）	10

行文文件 f 中每行的结束必须用 writeln(f) 明显地指示，而 writeln(output) 可以简写为 writeln. 如果一个行文文件要发送给打印机，那么每行不得多于 136 个字符. 每一行的第一个字符并不印刷，而被打印机解释为控制字符. 它们的含义如下所示:

'+'	不换行(叠印)
空白符	空一行
'0'	空二行
'1'	在打印前跳到下页的顶上

过程 writln (x) 用于标记文件 x 上一行的结束. 由于 CDC 操作系统关于行文文件表示上的规定，在某些情况下会迫使这个过程产生若干附加的空白符.因而在读行文文件时，行末可能包含若干并非明显书写的空白符. (对此很抱歉.)

13.3. 限　　制

1) **segmented** 是保留字.

2) 集合的基类型必须是下面的一种:

a) 至多 59 个元素的纯量类型(或它的子域类型);

b) 最小元素大于等于 0 且最大元素小于等于 58 的子域类型[1];

c) 字符类型的子域类型，其最大元素小于等于值 chr (58).

3) 标准的(内部的)过程或函数不能作为实在参数使用. 例如，为了使程序 11.6 能被 PASCAL 6000-3. 4 正确编译，应该象下面那样书写一些辅助函数:

```
...
```

1) 本条中两处"元素"后均应加"的序号". ——译者注

```
function sine(x: real) :real;
    begin sine := sin(x) end;

function zero(function f: real; a,b: real): real;
    begin ··· end;
      ···
begin
    read(x, y); writeln(x, y, zero(sine,x,y));
end.
```

4）不能构成文件的文件；但是允许文件作为数组和记录的元素.

5）串仅当长度小于 10 或为 10 的倍数时才能比较.

13.4. 附加的预定义类型、过程和函数

13.4.1. 附加的预定义类型

类型 **alfa**（α 类型）由

```
type alfa = packed array [1..10]of char;
```

预定义.（因而，α 类型的值实际上可以表示为一个字.）α 类型的常量是恰好由 10 个字符组成的串.

在 α 类型运算对象上可以应用的运算是赋值（:=）和比较，其中=与 <> 检查是否相等，而 <、<=、>= 与 > 则根据基本字符集来检查次序关系. α 值可以用过程 write 打印.

{程序 13.1
　α 值}

```
program egalfa(output);

var n1, n2: alfa;
```

```
begin write(' names:  ');
    n1: = 'raymond  '; n2 := 'debby
    if n2 < n1 then writeln(n2, n1)
               else writeln(n1, n2)
end.
```

```
names:  debby      raymond
```

注意：不能直接读 α 值；建议大家采用以下方法：

```
var buf: array [1..10] of char;
    a: alfa; i: integer;
    ...
    for i := 1 to 10 do read(buf[i]);
    pack(buf, 1, a) {实现 read(a)}
    ...
```

13.4.2. 附加的预定义过程和函数

13.4.2.1. 过程

date(a)	把当前日期赋给 α 类型变量 a.
halt	终止该程序的执行，并进行错后转贮.
linelimit(f,x)	f 是一个行文文件，x 是整数表达式. 它的作用是：当请求写到文件 f 的行数多于 x 时，终止该程序.
message(x)	串 x 写进日志文件.（因而，x 应至多包含 40 个字符.）
time(a)	把当前时间赋给 α 类型变量 a.

putseg、getseg 以及 rewrite 与 reset 的扩充已在 13.1.1. 节中讨论.

13.4.2.2. 函数

card(x)	等于集合 x 的基数（即集合 x 所含元素的个数）.
clock	无参函数. 结果为整数值，它等于该作业已耗费的中央处理机时间. 它以毫秒为单位.
expo(x)	结果为整数值，它等于实数 x 的浮点表示的阶. expo(x) = entier(log2(abs(x))).
undefined (x)	布尔函数. 当实数值 x "出界"或"不定的"时为真，否则为假[7].
eos(x)	(已在 13.1.1. 中讨论.)
trunc(x, n)	= trunc (x * y)，其中 n 是个整数表达式，且 y = 2 ** n.

14. 如何使用 PASCAL 6000-3.4 系统

14.1. 控制语句(对 SCOPE 3.4)

一个 PASCAL 作业通常分为四步. 第一步安装 PASCAL 编译程序. 第二步是编译，它产生源程序的列表和编译好的代码. 第三步把编译程序编译好的存放在辅助存贮器上的代码安装好，并把它与已预先编译好的输入输出处理例行程序联结起来. 这些例行程序是以"程序库"文件的方式提供的. 最后，执行已编译好且安装好的代码. 这四个步骤是通过**控制语句**形式向操作系统发适当的命令所启动的. 这些语句的实际形式以及它们的缩写形式（安装命令和执行命令通常是由单独一个语句发出的)完全依赖于所用的操作系统，并因此

必定是根据特定的装置来规定的.

程序首部的形式文件标识符所对应的实在文件参数，必须在用于启动执行编译好的程序的语句中指明（通常是由 EXECUTE 命令启动）.

编译程序本身也是个 PASCAL 程序. 它的首部是

program Pascal(input, output, lgo);

第一个形式参数所标记的文件是代表要编译的源程序;第二个是程序列表；第三个是编译好的“二进制的”可再定位代码.

CDC 操作系统允许省略控制语句中的实在参数. 如果一个实在文件名字省略了,那么 PASCAL 关于程序参数的约定规定,对应的形式文件标识符被用作实在文件名字. 因此，标准文件 INPUT、OUTPUT 和 LGO 分别自动被假定为 PASCAL 编译程序的源文件、列表和可再定位二进制代码的省缺文件. 可是注意,这三个文件当然亦可为其他文件,如果这些文件的名字用作实在参数的话. 注意：实在参数至多七个字符.

14.2. 编译程序的任选

可以通知编译程序根据某些任选的条件生成代码；特别是可以请求插入或删去运行时检查用的指令. 编译程序的伪指令以注释形式书写,并以 $ 字符标志;这个 $ 字符必须是注释的第一个字符:

{ $〈任选序列〉〈任何注释〉}

例: {$T+, P+}

这种任选序列是一个由逗号分隔的指令序列. 每条指令由一个字母后面跟一个加号 (+)、减号(−)或数字组成. 该

字母指示是哪个任选;加号表示该任选要起作用;减号表示不起作用.(数字见下面任选X与B.)

下面是目前可以使用的任选:

T 含有运行时检查,它检验

a) 所有数组的下标运算,以确保下标落在规定的数组界内;

b) 所有子域类型变量的赋值,使被赋值的一定落在规定的范围内;

c) 所有除法,以确保不以0作为除数;

d) 所有自动的整数-实数转换,以确保被转换的值满足

$$abs(i) <= maxint$$

e) 所有分情形语句,以确保分情形选择器的值对应于所规定的分情形标号中的某一个.

省缺值 = T +

P 生成某些必要的代码,以便在运行时出错情况下书写一个完全的错后转贮(见14.3.2.节).

省缺值=P+

X 如果X后面跟一个数字 n(0<=n<=6),那么前n个参数用寄存器X0 至 X(n-1)来传递(第一个在 X0,第二个在 X1,等等).否则用地址为 B6 + 3 至 B6 + n + 2 的单元传递.

在 n > 0 时,编译程序生成的代码长度会缩短,多半对代码也有所改进.但是程序员必须意识到: n > 0 时,编译程序不能利用寄存器 X0 到 $X_{min}(n-1, i-2)$ 来传递第 i 个参数.因此,有可能在 n > 0 时,编译程序给出"越出寄存器"这个信息; n = 0 时不会.

省缺值=X4

E 允许程序员控制他在程序中说明的目标代码模块(过程

与函数)入口点的符号. 约定如下:

——说明为 "extern" 或 "fortran" 的模块，其入口点名字是从过程标识符截取的前七个字符;

——局部模块入口点的名字是依赖于 E- 任选的当前值(在分析该模块的时刻):

E— 由编译程序生成的特殊符号

E+ 取模块名的前七个字符

每当截取模块名作入口点符号时 (E+以及 "extern" 或 "fortran" 时)，程序员有责任避免出现重复的入口点符号.

省缺值 =E—

L 控制程序行文的列表

省缺值 =L+

U 允许用户限制每个源行中有关字符的个数为72. 每行下余部分作为注释来处理. 当 U — 时，有关字符个数为120. 每行的下余部分仍作为注释来处理.

省缺值 =U—

B 用于规定文件缓冲器的下界. 如果 B 后面为数字 d (1<=d<=9),那么编译程序计算缓冲器大小 S 时,保证满足: S > 128 * d 个字.

省缺值 =B1

因为这些编译程序任选可以书写在程序中任何地方，故可以有选择地在程序的特定部分上使它们起作用.

14.3. 出 错 信 息

14.3.1. 编译程序的

编译程序用箭头指示所查出的错误. 箭头指向程序行文

中有关的地方，箭头后的数字表示附录E中的信息。

14.3.2. 运行时的（错后转贮）

当编译程序的任选P接通时（即P+），编译程序所生成的代码能在出现运行错误时印出可读的"转贮"。这种转贮包括下列信息：

1）陷井的原因和它出现的地方。

2）在陷井时正在活动的每个过程（函数）的描述。这些描述按照和它们的调用相反的次序出现，其内容如下：

a）过程的名字；

b）它调用的位置；

c）局部变量和参数的名字及值的表。

3）主程序中全程变量的值。

只列出类型为整数、实数、布尔和字符的变量和参数。指示字的值或是 **nil**，或是一个八进制值（地址）。对其它纯量类型的变量，印出它们当前值的序号。对于任何一个过程，任选P断开时(即P−)，转贮中只有过程的名字和它们调用的位置。

在递归调用的情况下，每个过程只列出最后（最近）三次出现。

参 考 文 献

[1] N. Wirth, The Programming Language PASCAL, *Acta Informatica*, 1, 35～63, 1971.

[2] ——, Program Development by Stepwise Refinement, *Comm. ACM* 14, 221～227, April 1971.

[3] ——, Systematic Programming, Prentice-Hall, Inc., 1973.

[4] O. J. Dahl, E. W. Dijkstra, C. A. R. Hoare, Structured Programming, Academic Press Inc., 1972.

[5] C. A. R. Hoare and N.Wirth, An Axiomatic Definition of the Programming Language Pascal, *Acta Informatica*, 2, 335～355, 1973.

[6] D. E. Knuth, THE ART OF COMPUTER PROGRAMMING, vol 1, Fundamental Algorithms, Addison-Wesley, 1968.
[7] SCOPE Reference Manual, CDC 6000 Version 3. 4. 1, Control Data Corporation, 1973.
[8] N. Wirth, The Design of a Pascal Compiler, *SOFTWARE-Practice and Experience*, **1**, 309~333, 1971.

附录 A 标准过程和标准函数

A.1. 文件处理的过程

put(f) 把缓冲器变量 f↑ 的值添加到文件 f 上。它仅当执行前 eof(f) 为真时可以使用。eof(f) 仍为真，f↑ 变成无定义。

get(f) 把当前文件位置移到下个成分上，并把该成分的值赋给缓冲器变量 f↑。如果不存在下个成分，那么 eof(f) 变成真，f↑ 的值无定义。该过程仅当执行前 eof(f) 为假时可以使用。

reset(f) 设置当前文件位置为该文件的开始以便读，也就是把 f 第一元素的值赋给缓冲器变量f↑。当 f 不空时，eof(f) 变为假；否则，f↑ 无定义，而且 eof(f) 仍为真。

rewrite(f) f 的当前值替换为空文件。eof(f)变为真，并可以写入一个新文件。

page(f) 指示打印机在打印行文文件 f 的下一行之前，跳到下一页的顶上。

read、readln、write 和 writeln 已在 12 章中讨论。

A.2. 动态分配的过程

new(p)	(设 p 是类型 T 的指示字变量.)分配一个(类型为 T 的)新变量 v，并把 v 的指示字引用值赋给指示字变量 p[1]. 如果 v 的类型是带变体的记录类型,那么形式
new(p,t1,…, tn)	能用于分配一个带标志字段值为 t1,…, tn 的变体的变量. 这些标志字段值必须按照它们说明的次序连续地列出，并且在执行期间不能改变它们.
dispose(p)	它指示由变量p↑所占用的存贮不再需要.
dispose(p,t1,…,tn)	它可用于指示由变量p↑(其标志字段值为 t1,…, tn) 所占用的存贮不再需要. 这些标志字段值必须等于该变量分配时的值.

A.3. 数据转换的过程

pack (a, i, z)

如果 a 是类型为

array [m··n] **of** T

的数组变量，z 是类型为

packed array[u··v] **of** T

的变量,其中 n − m>=v − u，那么该过程等价于

```
for j:= u to v do
z[j] := a[j − u + i]
```

1) 括号部分为译者所加. ——译者注

unpack (z, a, i)　等价于

```
for j := u to v do
a[j − u + 1] := z [j]
```

（在这两种情况下，j 标记不在该程序任何其他地方出现的辅助变量.）

A.4. 算 术 函 数

abs(x)　计算 x 的绝对值. 结果的类型与 x 的类型相同，必须是整数类型或实数类型.

sqr(x)　计算 x*x. 结果的类型与 x 的类型相同，必须是整数类型或实数类型.

sin(x)　下面几个函数中，x 的类型必须是实数类型或整数类型. 结果的类型总是实数类型.

cos(x)

arctan(x)

exp(x)

ln(x)　（自然对数）

sqrt(x)　（平方根）

A.5. 谓词(布尔函数)

odd(x)　x 的类型必须是整数类型；当 x 是奇数时，结果为真，否则为假.

eoln(f)　在读行文文件 f 时，如果到达当前行的结束符号，则结果为真，否则为假.

eof(f)　在读文件 f 时，如果到达“文件的结束符号”，则结果为真，否则为假.

A.6. 转 换 函 数

trunc(x)	x 必须属于实数类型;在 x>=0 时,结果为小于或等于 x 的最大整数;在 x < 0 时,结果为大于或等于 x 的最小整数.
round(x)	x 必须属于实数类型;结果属于整数类型,它是 x 的舍入值. 即 round(x) = trunc (x + 0.5), 对 x>=0 trunc (x − 0.5), 对 x < 0
ord(x)	变元 x 在其类型所定义的值集合里的序号.
chr(x)	x 必须属于整数类型,结果是序号为 x 的字符(如果它存在).

A.7. 其他标准函数

succ(x)	x 属于任意的纯量类型 (除实数类型外),结果是 x 的后继值(如果它存在).
pred(x)	x 属于任意的纯量类型 (除实数类型外),结果是 x 的前启值(如果它存在).

附录 B 运算符一览表

运算符		运算	运算对象类型	结果类型
:=		赋值	除文件类型外任何类型	-
算术运算符	+(单目)	恒等	整数或实数	同运算对象
	-(单目)	变符		
	+	加		整数或实数
	-	减		
	*	乘		
	div	整数除	整数	整数
	/	实数除	整数或实数	实数
	mod	取模	整数	整数
关系运算符	=	相等	纯量、串、集合、或指示字	布尔
	<>	不等		
	<	小于	纯量或串	
	>	大于		
	<=	小于或等于	纯量或串	
		集合包含	集合	
	>=	大于或等于	纯量或串	
		集合包含	集合	
	in	属于集合	第一运算对象为任何纯量类型，第二个是它的集合类型	
逻辑运算符	not	否定	布尔	布尔
	or	析取		
	and	合取		
集合运算符	+	并	任何集合类型T	T
	-	集合差		
	*	交		

附录 C　字定界符和标准标识符表[1)]

C.1. 标准标识符表

常量:

false, true, maxint

类型:

integer, Boolean, real, char, text

程序参数:

input, output

函数:

abs, arctan, chr, cos, eof, eoln, exp, ln, odd, ord, pred, round, sin, sqr, sqrt, succ, trunc

过程:

get, new, pack, page, put, read, readln, reset, rewrite, unpack, write, writeln

C.2. 字定界符(保留字)表

and	**array**	**begin**	**case**
const	**div**	**do**	**downto**
else	**end**	**file**	**for**
function	**goto**	**if**	**in**
label	**mod**	**nil**	**not**
of	**or**	**packed**	**procedure**
program	**record**	**repeat**	**set**
then	**to**	**type**	**until**

1) "表"字之前各字为译者所加。——译者注

var **while** **with**

C.3. PASCAL 6000-3.4 中非标准的预定义标识符

类型:

alfa

函数:

card, clock, eos, expo, undefined

过程:

date, getseg, halt, linelimit, message, potseg, time

附录D 语 法

D.1. 巴科斯-瑙尔范式(BNF)

注意: 下面的符号是属于 BNF 形式描述的元语言符号，不是 PASCAL 语言的符号.

:: = | { }

花括号标记它所括的符号可能重复 0 次或 n 次. 一般说来，

A :: = {B}

是下面纯递归规则的简短形式

A :: = 〈空〉 | AB

〈程序〉::=〈程序首部〉〈分程序〉.

〈程序首部〉 :: = **program**〈标识符〉(〈文件标识符〉
{,〈文件标识符〉});

〈文件标识符〉::=〈标识符〉

〈标识符〉::=〈字母〉{〈字母或数字〉}

〈字母或数字〉::=〈字母〉|〈数字〉

〈分程序〉::=〈标号说明部分〉〈常量定义部分〉
〈类型定义部分〉〈变量说明部分〉
〈过程与函数说明部分〉〈语句部分〉
〈标号说明部分〉::=〈空〉 | **label** 〈标号〉{,〈标号〉};
〈标号〉::=〈无符号整数〉
〈常量定义部分〉::=〈空〉 | **const** 〈常量定义〉
{;〈常量定义〉};
〈常量定义〉::=〈标识符〉=〈常量〉
〈常量〉::=〈无符号数〉|〈正负号〉〈无符号数〉|
〈常量标识符〉|〈正负号〉〈常量标识符〉|〈串〉
〈无符号数〉::=〈无符号整数〉|〈无符号实数〉
〈无符号整数〉::=〈数字〉{〈数字〉}
〈无符号实数〉::=〈无符号整数〉·〈数字〉{〈数字〉}|
〈无符号整数〉·〈数字〉{〈数字〉}
E〈标度因子〉|〈无符号整数〉E
〈标度因子〉
〈标度因子〉::=〈无符号整数〉|〈正负号〉
〈无符号整数〉
〈正负号〉::= + | −
〈常量标识符〉::=〈标识符〉
〈串〉::='〈字符〉{〈字符〉}'
〈类型定义部分〉::=〈空〉| **type** 〈类型定义〉
{;〈类型定义〉};
〈类型定义〉::=〈标识符〉=〈类型〉
〈类型〉::=〈简单类型〉|〈构造类型〉|〈指示字类型〉
〈简单类型〉::=〈纯量类型〉|〈子域类型〉|
〈类型标识符〉
〈纯量类型〉::=(〈标识符〉{,〈标识符〉})

〈子域类型〉::=〈常量〉..〈常量〉

〈类型标识符〉::=〈标识符〉

〈构造类型〉::=〈非紧缩构造类型〉|
packed〈非紧缩构造类型〉

〈非紧缩构造类型〉::=〈数组类型〉|〈记录类型〉|
〈集合类型〉|〈文件类型〉

〈数组类型〉::= **array** [〈下标类型〉{,〈下标类型〉}]
of〈成分类型〉

〈下标类型〉::=〈简单类型〉

〈成分类型〉::=〈类型〉

〈记录类型〉::= **record**〈字段表〉**end**

〈字段表〉::=〈固定部分〉|〈固定部分〉;
〈变体部分〉|〈变体部分〉

〈固定部分〉::=〈记录段〉{;〈记录段〉}

〈记录段〉::=〈字段标识符〉{,〈字段标识符〉}:
〈类型〉|〈空〉

〈变体部分〉::= **case**〈标志字段〉〈类型标识符〉**of**
〈变体〉{;〈变体〉}

〈标志字段〉::=〈字段标识符〉:|〈空〉

〈变体〉::=〈分情形标号表〉:(〈字段表〉)|〈空〉

〈分情形标号表〉::=〈分情形标号〉{,〈分情形标号〉}

〈分情形标号〉::=〈常量〉

〈集合类型〉::= **set of**〈基类型〉

〈基类型〉::=〈简单类型〉

〈文件类型〉::= **file of**〈类型〉

〈指示字类型〉::=↑〈类型标识符〉

〈变量说明部分〉::=〈空〉| **var**〈变量说明〉{;
〈变量说明〉};

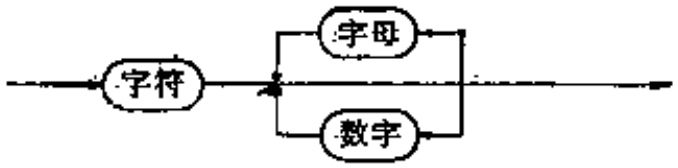
〈变量说明〉::=〈标识符〉{,〈标识符〉}:〈类型〉
〈过程与函数说明部分〉::={〈过程或函数说明〉;}
〈过程或函数说明〉::=〈过程说明〉|〈函数说明〉
〈过程说明〉::=〈过程首部〉〈分程序〉
〈过程首部〉::= **procedure** 〈标识符〉; | **procedure**
〈标识符〉(〈形式参数段〉{;
〈形式参数段〉});
〈形式参数段〉::=〈参数组〉|**var** 〈参数组〉|
function 〈参数组〉|
procedure 〈标识符〉{,〈标识符〉}
〈参数组〉::=〈标识符〉{,〈标识符〉}:〈类型标识符〉
〈函数说明〉::=〈函数首部〉〈分程序〉
〈函数首部〉::= **function** 〈标识符〉:〈结果类型〉; |
function 〈标识符〉(〈形式参数段〉
{;〈形式参数段〉}):〈结果类型〉;
〈结果类型〉::=〈类型标识符〉
〈语句部分〉::=〈复合语句〉
〈语句〉::=〈无标号语句〉|〈标号〉:〈无标号语句〉
〈无标号语句〉::=〈简单语句〉|〈构造语句〉
〈简单语句〉::=〈赋值语句〉|〈过程语句〉|
〈转向语句〉|〈空语句〉
〈赋值语句〉::=〈变量〉:=〈表达式〉|〈函数标识符〉
:=〈表达式〉
〈变量〉::=〈整体变量〉|〈成分变量〉|〈引用变量〉
〈整体变量〉::=〈变量标识符〉
〈变量标识符〉::=〈标识符〉
〈成分变量〉::=〈下标变量〉|〈字段命名符〉|
〈文件缓冲器〉

〈下标变量〉::=〈数组变量〉[〈表达式〉
{,〈表达式〉}]
〈数组变量〉::=〈变量〉
〈字段命名符〉::=〈记录变量〉·〈字段标识符〉
〈记录变量〉::=〈变量〉
〈字段标识符〉::=〈标识符〉
〈文件缓冲器〉::=〈文件变量〉↑
〈文件变量〉::=〈变量〉
〈引用变量〉::=〈指示字变量〉↑
〈指示字变量〉::=〈变量〉
〈表达式〉::=〈简单表达式〉|〈简单表达式〉
〈关系运算符〉〈简单表达式〉
〈关系运算符〉::=＝|<>|<|<=|>=|>| **in**
〈简单表达式〉::=〈项〉|〈正负号〉〈项〉|
〈简单表达式〉〈加型运算符〉〈项〉
〈加型运算符〉::=＋|－| **or**
〈项〉::=〈因子〉|〈项〉〈乘型运算符〉〈因子〉
〈乘型运算符〉::= ＊ | / | **div** | **mod** | **and**
〈因子〉::=〈变量〉|〈无符号常量〉|(〈表达式〉)|
〈函数命名符〉|〈集合〉| **not** 〈因子〉
〈无符号常量〉::=〈无符号数〉|〈串〉|
〈常量标识符〉| **nil**
〈函数命名符〉::=〈函数标识符〉|〈函数标识符〉
(〈实在参数〉{,〈实在参数〉})
〈函数标识符〉::=〈标识符〉
〈集合〉::=[〈元素表〉]
〈元素表〉::=〈元素〉{,〈元素〉}|〈空〉
〈元素〉::=〈表达式〉|〈表达式〉..〈表达式〉

〈过程语句〉::=〈过程标识符〉|〈过程标识符〉
('〈实在参数〉{,〈实在参数〉})
〈过程标识符〉::=〈标识符〉
〈实在参数〉::=〈表达式〉|〈变量〉|〈过程标识符〉|
〈函数标识符〉
〈转向语句〉::= **goto** 〈标号〉
〈空语句〉::=〈空〉
〈空〉::=
〈构造语句〉::=〈复合语句〉|〈条件语句〉|〈重复语句〉|
〈开域语句〉
〈复合语句〉::= **begin** 〈语句〉{;〈语句〉} **end**
〈条件语句〉::=〈如果语句〉|〈分情形语句〉
〈如果语句〉::= **if** 〈表达式〉 **then** 〈语句〉|
if 〈表达式〉 **then** 〈语句〉 **else** 〈语句〉
〈分情形语句〉::= **case** 〈表达式〉 **of** 〈分情形表元素〉
{;〈分情形表元素〉} **end**
〈分情形表元素〉::=〈分情形标号表〉:〈语句〉|〈空〉
〈分情形标号表〉::=〈分情形标号〉{,〈分情形标号〉}
〈重复语句〉::=〈当语句〉|〈直到语句〉|〈循环语句〉
〈当语句〉::= **while** 〈表达式〉 **do** 〈语句〉
〈直到语句〉::= **repeat** 〈语句〉{;〈语句〉} **until** 〈表达式〉
〈循环语句〉::= **for** 〈控制变量〉:=〈循环表〉 **do** 〈语句〉
〈循环表〉::=〈初值〉 **to** 〈终值〉|〈初值〉 **downto** 〈终值〉
〈控制变量〉::=〈标识符〉
〈初值〉::=〈表达式〉
〈终值〉::=〈表达式〉
〈开域语句〉::= **with** 〈记录变量表〉 **do** 〈语句〉
〈记录变量表〉::=〈记录变量〉{,〈记录变量〉}

D. 2. PASCAL 语法图

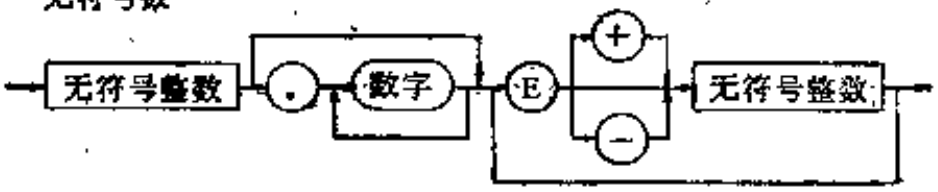
标识符



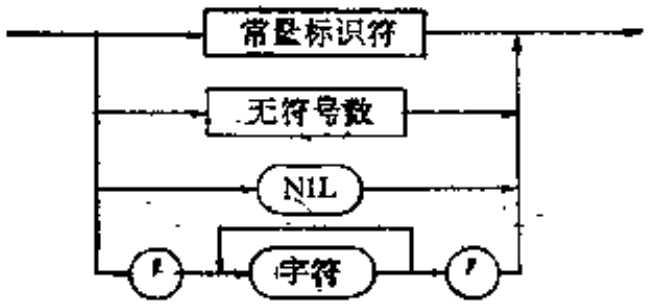
无符号整数



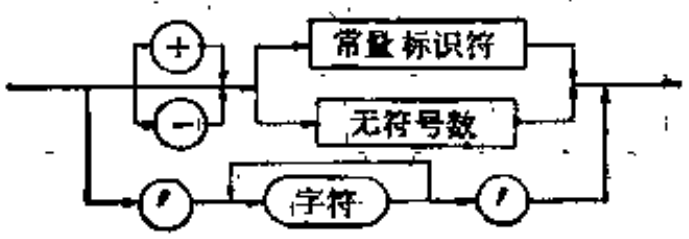
无符号数



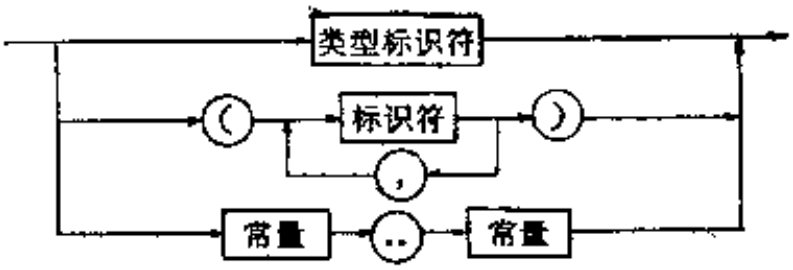
无符号常量



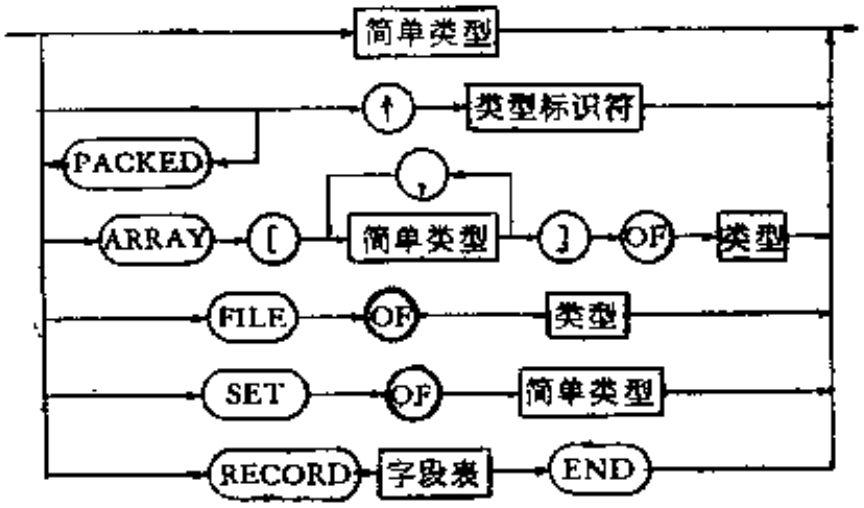
常量



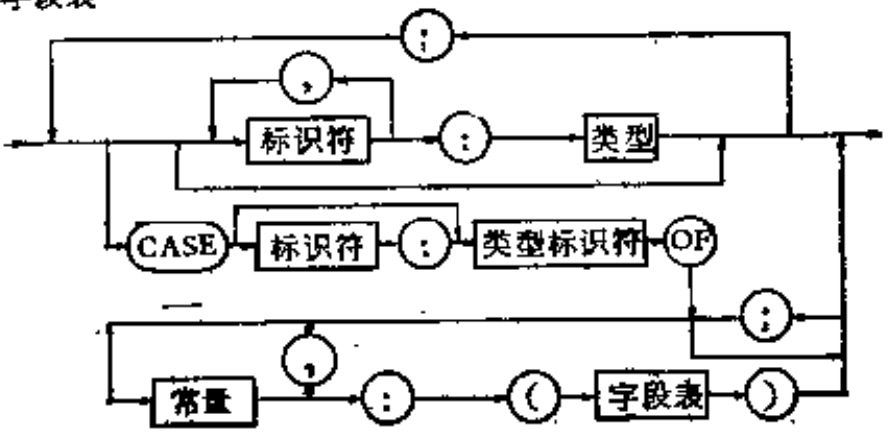
简单类型


类型标识符
(
标识符
)
,
常量
..
常量

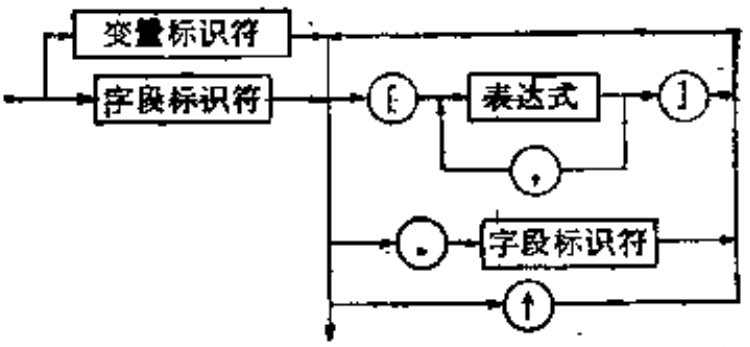
类型


简单类型
↑
类型标识符
PACKED
,
ARRAY
[
简单类型
]
OF
类型
FILE
OF
类型
SET
OF
简单类型
RECORD
字段表
END

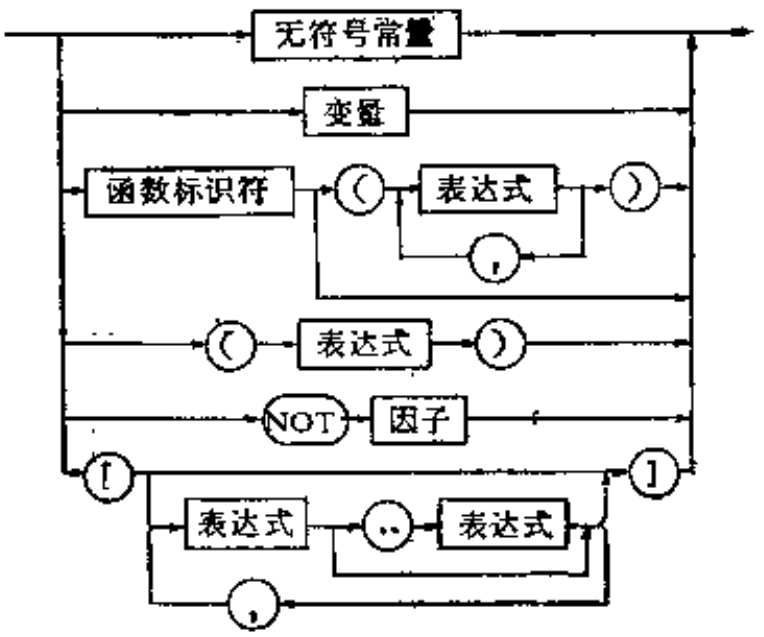
字段表


;
,
标识符
:
类型
CASE
标识符
:
类型标识符
OF
;
,
常量
:
(
字段表
)

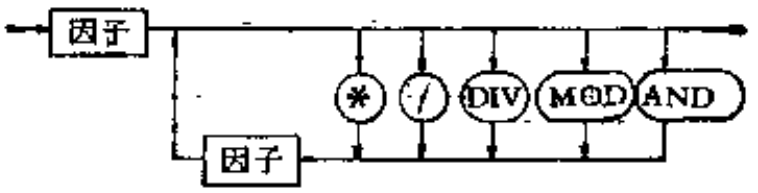
变量


变量标识符
字段标识符
[
表达式
,
]
.
字段标识符
↑

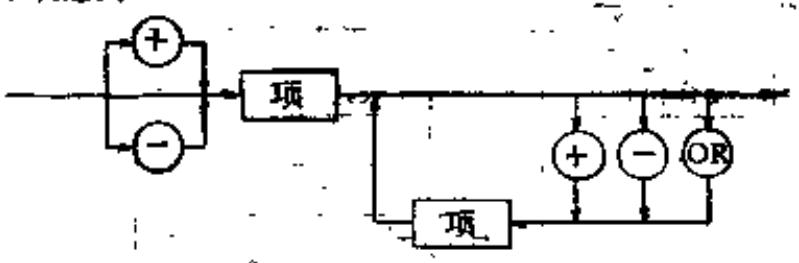
因子


无符号常量
变量
函数标识符
(
表达式
,
)
(
表达式
)
NOT
因子
[
表达式
..
表达式
,
]

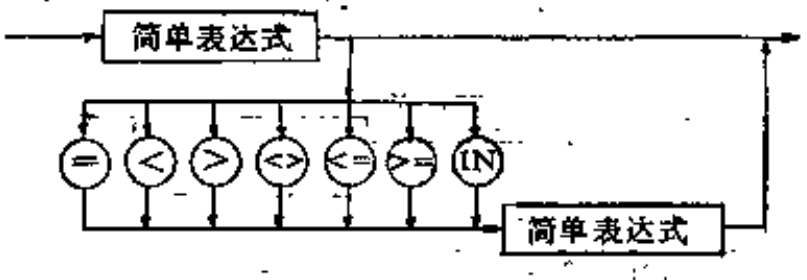
项


因子
*
/
DIV
MOD
AND
因子

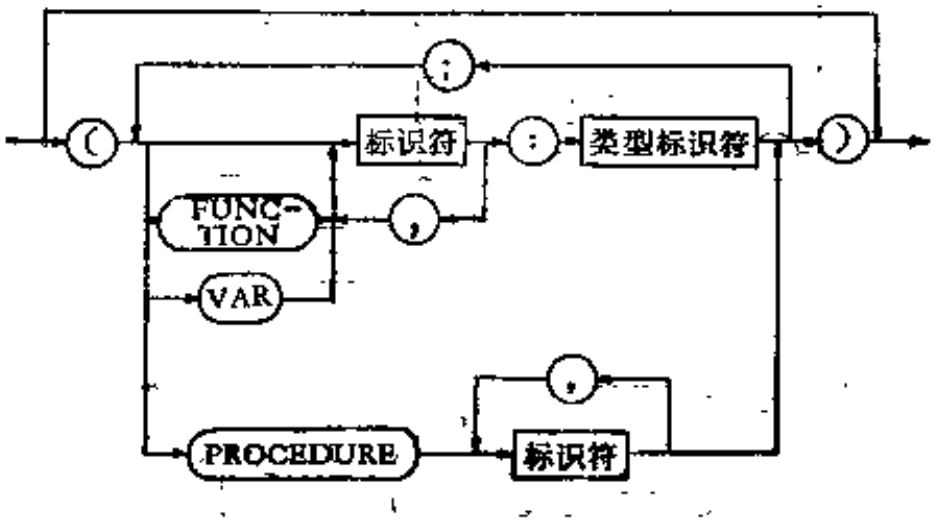
简单表达式

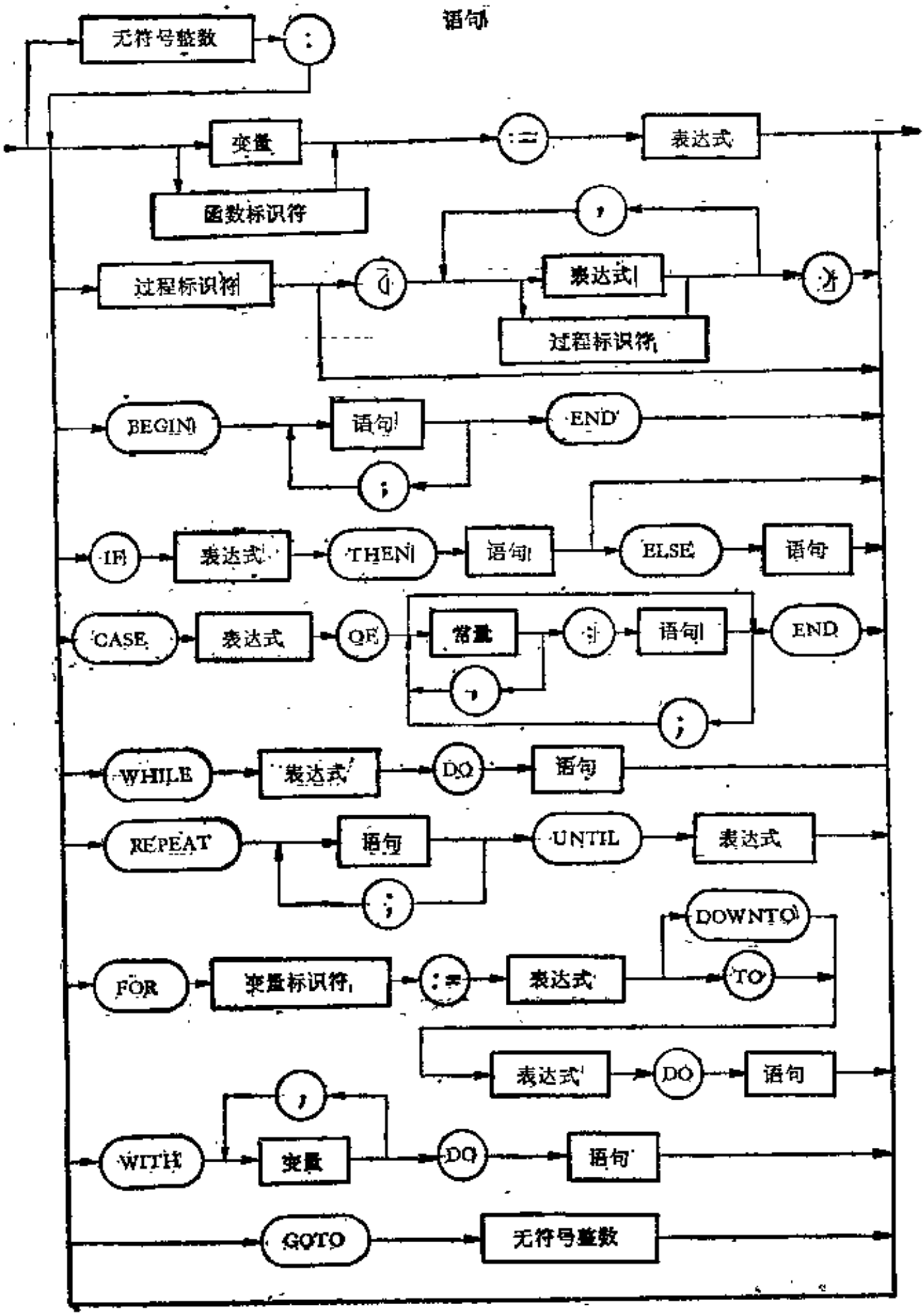


表达式

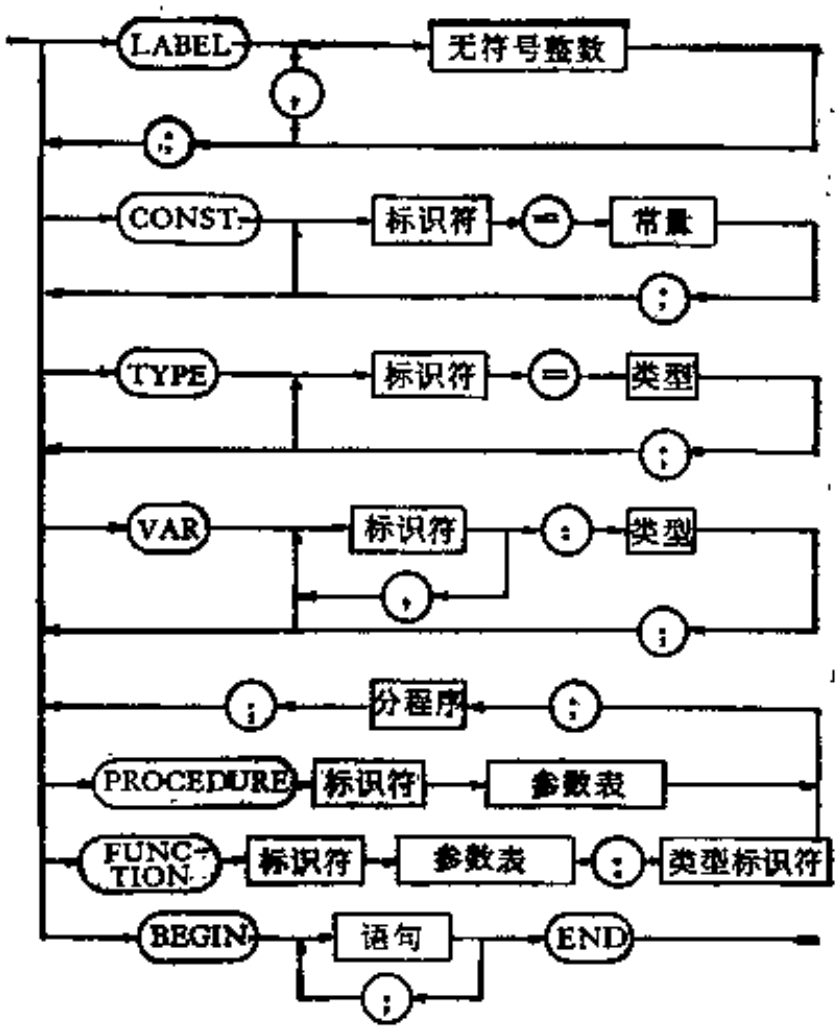


参数表

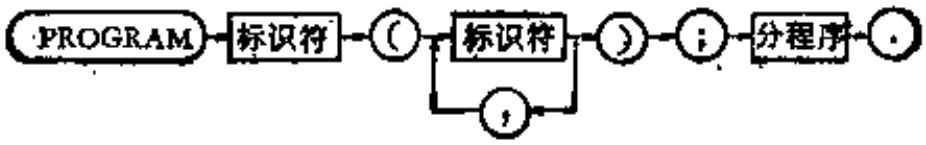



语句
无符号整数
:
变量
:=
表达式
函数标识符
过程标识符
(
,
)
BEGIN
语句
END
;
IF
表达式
THEN
ELSE
CASE
OF
常量
:
WHILE
DO
REPEAT
UNTIL
FOR
变量标识符
DOWNTO
TO
WITH
GOTO
无符号整数

分程序


LABEL
无符号整数
,
;
CONST
标识符
=
常量
;
TYPE
标识符
=
类型
;
VAR
标识符
:
类型
,
;
;
分程序
;
PROCEDURE
标识符
参数表
FUNC-TION
标识符
参数表
:
类型标识符
BEGIN
语句
END
;

程序


PROGRAM
标识符
(
标识符
)
;
分程序
.
,

附录 E　出错号一览表

1：简单类型里出错
2：期望标识符
3：期望 'program'
4：期望 ')'
5：期望 ':'
6：不合法符号
7：参数表里出错
8：期望 'of'
9：期望 '('
10：类型里出错
11：期望 '['
12：期望 ']'
13：期望 'end'
14：期望 ';'
15：期望整数
16：期望 '='
17：期望 'begin'
18：说明部分里出错
19：字段表里出错
20：期望 '.'
21：期望 '*'

50：常量里出错
51：期望 ':='
52：期望 'then'

53: 期望 'until'
54: 期望 'do'
55: 期望 'to'/'downto'
56: 期望 'if'
57: 期望 'file'
58: 因子里出错
59: 变量里出错

101: 标识符说明两次
102: 下界超过上界
103: 标识符类别不适当
104: 标识符未说明
105: 不允许正负号
106: 期望数
107: 不相容的子域类型
108: 这里不允许文件
109: 类型不可为实数
110: 标志字段类型必须是纯量类型或子域类型
111: 与标志字段的类型不相容
112: 下标类型不可为实数
113: 下标类型必须是纯量类型或子域类型
114: 基类型不可为实数
115: 基类型必须是纯量类型或子域类型
116: 标准过程参数的类型里出错
117: 向前引用不符合要求
118: 向前引用类型标识符在变量说明中
119: 已向前说明;参数表不允许重复
120: 函数的结果类型必须是纯量、子域或指示字类型

121：不允许文件值参数
122：函数已向前说明；结果类型不允许重复
123：函数说明里漏了结果类型
124：F-格式仅对实数
125：标准函数参数的类型里出错
126：参数的数目与说明不一致
127：不合法的参数替代
128：参数函数的结果类型与说明不一致
129：运算对象类型冲突
130：表达式不属于集合类型
131：只允许检查相等性
132：不允许真包含
133：文件不允许比较
134：运算对象的类型不合法
135：运算对象的类型必须是布尔类型
136：集合元素的类型必须是纯量类型或子域类型
137：集合各元素的类型不相容
138：变量的类型不是数组类型
139：下标类型与说明不相容
140：变量的类型不是记录类型
141：变量的类型必须是文件类型或指示字类型
142：不合法的参数替代
143：循环控制变量的类型不合法
144：表达式的类型不合法
145：类型冲突
146：文件不允许赋值
147：标号类型与选择表达式不相容
148：子域类型的界必须是纯量

149: 下标类型不可为整数
150: 不允许对标准函数赋值
151: 不允许对形式函数赋值
152: 这个记录中没有这样的字段
153: read 中类型出错
154: 实在参数必须是个变量
155: 控制变量不可在中间层次上说明
156: 分情形标号多重定义
157: 分情形语句中所分的情形太多
158: 遗漏相应的变体说明
159: 标志字段不允许实数或串
160: 原先的说明不是向前说明
161: 再次向前说明
162: 参数大小必须是常量
163: 说明里遗漏变体
164: 标准过程或标准函数不允许替代
165: 标号多重定义
166: 标号多重说明
167: 标号未说明
168: 标号未定义
169: 基集合里出错
170: 期望值参数
171: 标准文件又被说明
172: 外部文件未说明
173: 期望 FORTRAN 过程或函数
174: 期望 PASCAL 过程或函数
175: 程序首部里遗漏文件 "input"
176: 程序首部里遗漏文件 "output"

177：这里不允许对函数标识符赋值
178：记录变体多重定义
179：实在过程/函数的 X-任选与形式说明不匹配
180：控制变量不可为形式参数
181：地址的常量部分越界

201：实数常量中出错：期望数字
202：串常量不可超出源行
203：整数常量越界
204：八进数里出现 8 或 9
205：不允许 0 串
206：实数常量的整数部分越界

250：标识符的作用域嵌套层次太多
251：过程与/或函数嵌套层次太多
252：过程入口向前引用太多
253：过程太长
254：这个过程里长常量太多
255：这个源行错误太多
256：外部引用太多
257：external 过多
258：局部文件太多
259：表达式太复杂
260：出口标号太多

300：除数为 0
301：对该值没有提供分情形
302：下达表达式越界

303：要赋的值越界

304：元素表达式越界

398：实现的限制

399：可变维数的数组未实现

附录F　程序设计实例

{这两个读写实数的过程，摘自于标准过程 read (f, x) 和 write (f, x:n)}

```
procedure rdr(var f: text; var x: real);
    {读"自由格式"的实数}
    const t48 = 281474976710656;
        limit = 56294995342131;
        z = 27; {ord ('0')}
        lim1 = 322;     {最大指数}
        lim2 = -292;    {最小指数}
    type posint = 0..323;
    var ch: char; y: real; a,i,e: integer;
        s,ss: boolean;  {正负号}
    function ten(e: posint): real; {=10**e, 0<e<322}
        var i: integer; t: real;
    begin i := 0; t := 1.0;
        repeat if odd(e) then
            case i of
                0: t := t * 1.0e1;
                1: t := t * 1.0e2;
                2: t := t * 1.0e4;
                3: t := t * 1.0e8;
                4: t := t * 1.0e16;
                5: t := t * 1.0e32;
                6: t := t * 1.0e64;
                7: t := t * 1.0e128;
                8: t := t * 1.0e256
```

```
            end;
            e:=e div 2; i:=i+1
        until e=0;
        ten:= t
    end;

begin
    {跳过数前面的空白符}
        while f↑=' ' do get(f);
        ch:=f↑;
        if ch='-' then
            begin s:=true; get(f); ch:=f↑
            end else
            begin s:=false;
                if ch='+' then
                begin get(f); ch:=f↑
                end
            end;
        if not (ch in ['0'..'9']) then
        begin message('** digit expected'); halt;
        end;
        a:=0; e:=0;
        repeat if a<limit then a:=10*a+ord(ch)-z
                          else e:=e+1;
            get(f); ch:=f↑
        until not (ch in ['0'..'9']);
        if ch='.' then
        begin {读小数部分} get(f); ch:=f↑;
            while ch in ['0'..'9'] do
            begin if a<limit then
                begin a:=10*a+ord(ch)-z; e:=e-1
                end;
                get(f); ch:=f↑
            end
        end;
        if ch='e' then
        begin {读标度因子} get(f); ch:=f↑;
```

```
            i := 0;
            if ch = '-' then
            begin ss := true; get(f); ch := f↑
            end else
            begin ss := false; if ch = '+' then
                begin get(f); ch := f↑
                end
            end;
            if ch in ['0'..'9'] then
            begin i := ord(ch) - z; get (f); ch := f↑
                while ch in ['0'..'9'] do
                begin if i<limit then i := 10 * i + ord(ch) - z;
                    get(f); ch := f↑
                end
            end else
            begin message('digit expected'); halt
            end;
            if ss then e := e - i else e := e + i;
        end;
        if e < lim2 then
            begin a := 0; e: = 0
            end else
        if e > lim1 then
        begin message('**number too large'); halt end;
        {0 < a < 2**49}
        if a >= t48 then y := ((a + 1) div 2) * 2.0
                    else y := a;
        if s then y := -y;
        if e < 0 then x := y/ten(-e) else
        if e <> 0 then x := y * ten(e) else x := y;
end;

procedure wre(var f: text; x: real; n: integer);
    {按十进制浮点形式书写实数 x, 它占 n 个字符位置}
    {下面的常量是根据 CDC 浮点格式制定的}
    const t48 = 281474976710656; {=2**48}
```

```
        z = 27; {ord('ϕ')}
    type posint = ϕ..323;
    var c, d, e, eϕ, e1, e2, i: integer;

    function ten(e: posint): real; {10 **e, ϕ < e < 322}
        var i: integer; t: real;
    begin i := ϕ; t := 1.ϕ;
        repeat if odd(e) then
            case i of
                0: t := t * 1.ϕe1;
                1: t := t * 1.ϕe2;
                2: t := t * 1.ϕe4;
                3: t := t * 1.ϕe8;
                4: t := t * 1.ϕe16;
                5: t := t * 1.ϕe32;
                6: t := t * 1.ϕe64;
                7: t := t * 1.ϕe128;
                8: t := t * 1.ϕe256
            end;
            e:=e div 2; i:=i + 1
        until e = ϕ;
        ten: = t
    end {ten};

begin {至少需要 10 个字符:  b + 9.9e + 999}
    if undefined(x) then
    begin repeat f↑ := ' '; put(f); n := n - 1
          until n < = 1;
        f↑ := 'u'; put(f)
    end else
    if x = ϕ then
    begin repeat f↑ :=' '; put (f); n:=n - 1
          until n < = 1;
        f↑ := 'ϕ'; put(f)
    end else
    begin
        if n < = 1ϕ then n := 3 else n := n - 7;
```

```
repeat f↑ := ' '; put (f); n := n - 1
until n <= 15;
{1 < n <= 15, 要印的数字个数}
begin {检查数的符号,再取指数}
    if x < ϕ then
        begin f↑ := ' - '; put(f); x := -x
        end else begin f↑ := ' '; put(f) end;
    e := expo (x);
    if e >= ϕ then
        begin e := e*77 div 256 + 1; x := x/ten(e);
            if x >= 1.ϕ then
                begin x := x/1ϕ. ϕ; e := e + 1
                end
        end else
        begin e := (e + 1)*77 div 256; x := ten(-e) * x;
            if x < ϕ.1 then
                begin x := 1ϕ.ϕ * x; e := e - 1
                end
        end;
    {ϕ.1 <= x < 1.ϕ}
    case n of {舍入}
         2: x := x + ϕ.5e - 2;
         3: x := x + ϕ.5e - 3;
         4: x := x + ϕ.5e - 4;
         5: x := x + ϕ.5e - 5;
         6: x := x + ϕ.5e - 6;
         7: x := x + ϕ.5e - 7;
         8: x := x + ϕ.5e - 8;
         9: x := x + ϕ.5e - 9;
        1ϕ: x := x + ϕ.5e - 1ϕ;
        11: x := x + ϕ.5e - 11;
        12: x := x + ϕ.5e - 12;
        13: x := x + ϕ.5e - 13;
        14: x := x + ϕ.5e - 14;
        15: x := x + ϕ.5e - 15
    end;
    if x>= 1.ϕ then
```

```
                begin x := x * ϕ.1; e := e + 1;
                end;
            c := trunc(x, 48);
            c := 1ϕ * c; d := c div t48;
            f↑ := chr(d + z); put(f);
            f↑ := '.'; put(f);
            for i := 2 to n do
            begin c := (c - d * t48) * 1ϕ; d: = c div t48;
                f↑ := chr(d + z); put(f)
            end;
            f↑ := 'e'; put (f); e := e - 1;
            if e < ϕ then
                begin f↑ := '-'; put (f); e := - e;
                end else begin f↑ := '+'; put(f) end;
            e1 := e * 2ϕ5 div 2048; e2 := e - 1ϕ * e1;
            eϕ := e1 * 2ϕ5 div 2048; e1 := e1 - 1ϕ * eϕ;
            f↑ := chr(eϕ + z); put(f);
            f↑ := chr(e1 + z); put(f);
            f↑ := chr(e2 + z); put(f)
        end
    end
end {wre};
```

索　　引

本索引以汉语笔画和英文字母为序，各词条后面为章节号、图号、或程序号。

α (alfa, PASCAL 6000-3.4)　13.4.1.
input (标准文件)　9.2.
output (标准文件)　9.2.
PASCAL 6000-3.4　13., 14.
read (标准过程)　12.1.
write (标准过程)　12.2.
　(PASCAL 6000-3.4)　13.2.4.

二　　画

二叉树 (binary tree) 11.1

四　　画

文件 (file) 9.
,分段 (segmented) 13.1.1.
,外部 (external, PASCAL 6000-3.4) 13.2.1.
,行文 (text) 9.1.
—表示 (representation, PASCAL 6000-3.4) 13.2.2.
—类型 (type) 9.
巴科斯范式定义 (BNF definitions) 附录 D.1.
分程序 (block) 0.
分隔符 (separator) 1.

六　　画

字定界符表 (word-delimiter, table of) 附录 C.
字段表 (field list) 7.
字符集 (character set) 13.2.3.
过程 (procedure) 11.1.
,外部 (external, PASCAL 6000-3.4) 13.1.2.
—说明部分 (declaration part) 3.6.
—语句 (statement) 11.1.
—首部 (heading) 11.1.
,标准 (standard) 附录 A.
,预定义的 (predefined, PASCAL 6000-3.4.) 13.4.2.
先行设备 (lookahead device) 9.
名字优先顺序 (name precedence) 11.1.
向前引用 (forward reference) 11.3.
行结束符 (end of lines) 9.1.

七　　画

词汇表 (vocabulary) 1.
运行时出错信息 (run-time error messages) 14.3.2.

运算符 (operator) 4.1.
——一览表 (, summary of) 附录 B.
,加型 (adding) 2.1.,4.1.
,关系 (relational) 2.1.,4.1.
—优先顺序 (precedence) 4.1.
,乘型 (multiplaying) 2.1.,4.1.
,集合 (set) 8.
串 (string) 1.,6.
作用域 (scope) 0.

八　　画

注释 (comment) 1.
变量 (variable) 3.5.
,全程 (global) 11.1.
,局部 (local) 11.1.
—说明部分 (declaration part) 3.5.
,控制 (control) 4.3.3.
表,链表 (list, linked list) 10.
表示法 (notation) 1.
表达式 (expression) 4.1.
表格 (table)
分程序结构 (blok structure) 0.
打印机控制字符 (print control character) 9.2.,13.2.4.
字段宽度省缺值 (default value for f eld width) 13.2.4.
行文文件上的运算 (operation on textfile) 9.1.
特殊符号 (special symbol) 1.
函数 (function) 11.2.
—命名符 (designator) 11.2.
—说明部分 (declaration part) 3.6.
—首部 (heading) 11.2.
,标准 (standard) 附录 A.
,预定义的 (predefined, PASCAL 6000-3.4) 13.4.2.
限制 (restrictions, PASCAL 6000-3.4) 13.3
图 (figure)
ASCII 字符集 (按 CDC 次序) 图 13.a
CDC 科学字符集 (64 字符) 图 13.a

二叉树结构 图 11.b
无符号数 图 1.b
分程序结构 图 0.b
两个模特儿人 图 7.a
定义一个程序总结构的语法图 图 0.a
表达式 图 11.a
标识符 图 1.a
标准符号的另一种表示 图 13.b
插入后 图 10.c
插入前 图 10.b
链表 图 10.a
图式 (schemata)
从 "input" 读一行文 9.1.
从一行文文件读任意多个数据项 12.1.
写一行文 9.1.
写分段文件 13.1.1.
在 "output" 上写一行文 9.1.
把行文 x 复制到行文 y 上 9.1.
读一行文 9.1.
读分段文件 13.1.1.
读写分段文件 13.1.1.
参数 (parameter) 11.1., 11.2.
,形式 (formal) 11.1., 11.2.
,实在 (actual) 11.1., 11.2.

九　　画

说明部分 (declaration part) 3.
语句 (statement) 4.
,开域 (with) 7.1.
,分情形 (case) 4.4.2.
,过程 (procedure) 11.1.
,当 (while) 4.3.1.
,如果 (if) 4.4.1.
,条件 (conditional) 4.4.
,空 (empty) 4.2.
,直到 (repeat) 4.3.2.

,转向 (goto) 4.5.
,重复 (repetitive) 4.3.
,复合 (compound) 4.2.
,赋值 (assignment) 4.1.
,循环 (for) 4.3.3.
语法图 (syntax diagrams) 附录 D. 2.
类型 (type) 2.
,子域 (subrange) 5.2.
,文件 (file) 9.
,记录 (record) 7.
,布尔 (Boolean) 2.1.
,字符 (char) 2.4.
,纯量 (scalar) 5.1.
,实数 (real) 2.3.
—定义部分 (definition part) 3.4.
,标准 (standard, PASCAL 6000-3.4) 13.2.3.
,指示字 (pointer) 10.
,预定义的 (predefined, PASCAL 6000-3.4) 13.4.1.
,集合 (set) 8.
,数组 (array) 6.
,数据 (data) 2.
,整数 (integer) 2.2.
标号 (label) 3.2.
,分情形 (case) 7.1.
,转向 (go to) 3.2.,4.5.
—说明部分 (declaration part) 3.2.
标识符 (identifier) 1.
,标准 (standard) 附录 C.
保留字 (reserved words) 附录 C.

十 画

真值 (truth values) 2.1.
紧缩结构 (packed structure) 6.

十 一 画

控制语句 (control statement, PASCAL 6000-3.4) 14.1.

副作用 (side effect) 11.2.
常量定义部分 (constant definition part) 3.3.

十 二 画

等价 (equivalence) 2.1.
程序 (program) 1.
　一首部 (heading) 3.1.
　　(PASCAL 6000-3.4) 13.2.1.
程序和基本程序 (program and program part)
　beginend (复合语句) 程序 4.1
　bisect (二分法) 程序 11.6
　complex (复数运算) 程序 7.1
　convert (变换) 程序 3.1
　cosine (余弦) 程序 4.5
　egalfa (α 类型例子) 程序 13.1
　egfor (循环语句例子) 程序 4.4
　egrepeat (直到语句例子) 程序 4.3
　egwhile (当语句例子) 程序 4.2
　exponentiation (取自然指数的幂) 程序 4.8
　expon2 (取自然指数的幂) 程序 11.8
　fcount (频率计算) 程序 9.1
　graph1 (画函数曲线 1) 程序 4.9
　graph2 (画函数曲线 2) 程序 6.2
　inflation (贬值率) 程序 0.1
　insert (插入前端空白) 程序 9.2
　matrixmul (矩阵乘) 程序 6.3
　minmax (求最小值和最大值) 程序 6.1
　minmax2 (求最小值和最大值) 程序 11.1
　minmax3 (求最小值和最大值) 程序 11.2
　parameters (参数表) 程序 11.3
　postfix (化为后缀形式) 程序 11.4
　primes (找素数) 程序 8.2
　recursivegcd (递归方式求最大公约数) 程序 11.9
　roman (写罗马数字) 程序 4.7
　setop (集合运算) 程序 8.1
　sideffect (副作用) 程序 11.7

summming (求和) 程序 4.6
traversal (遍历) 程序 11.5
合并两个文件 9.
向前引用 11.3.
实数文件求和 9.
转向语句的例子 4.5.
指示字的例子 10.
编译程序 (compiler)
—出错信息 (error message) 14.3.1. 附录E.
—任选 (option, PASCAL 6000-3.4) 13.2.

十 三 画

数 (number) 1.
输入/输出 (I/O) 10.

十 五 画

蕴涵 (implication) 2.1.

第三部分　PASCAL 编译程序的设计

N. 沃　恩

0. 引　言

程序设计语言 PASCAL[1][1] 是按照下列主要目标设计和实现的：

1）寻找一种表达程序设计的基本概念与结构的表示方法，而且这种表达方式应该是系统、精确、合理的.

2）这种表示方法应该考虑到在程序研制的系统方法方面的新见解[2,3].

3）证明一个具有丰富灵活的数据构造能力和程序构造

1）本文写于 1971 年 7 月. 在此之后作者对该语言作了些修改，于 1974 年产生了“修订报告”(与本书第一部分“报告”相差甚微)，即所谓标准 PASCAL 语言.作者说，这主要出自对程序可移植性及独立于机器的可定义性这两点考虑. 它们的差别是

(1) 常量参数改为象 ALGOL 60 那样的值参数；

(2) 取消 class (类)结构，指示字变量从受囿于 class 变量改为受囿于任意的数据类型；

(3) 改变文件的处理，使得在 eof (f) 非真时缓冲器变量 f↑ 总有确定的值；

(4) 引入紧缩数组与紧缩记录，类型 alfa 变量作为紧缩字符数组的特例，这种改变也稍稍影响到字符串的表示方法；

(5) 增加程序首部并以外部文件作为参数；

(6) 所有标号都要说明；

(7) 稍稍改动了语法，如势集类型改为集合类型等；

(8) 增加了 readln、writeln 与 eoln 等标准过程；

(9) 语言用 ISO 字符集定义. ——译者注

能力的语言可以由一个中等规模的编译程序高效率地实现.

4）证明一个编译程序，如果采用一种具有灵活的数据结构和程序结构的、独立于机器的语言来描述，会增加它的可读性与可证明性，因而还会提高它的可靠性；并证明这并不需要以丧失某种效率为代价.

5）透彻地了解组织大程序和管理软件工作方面的方法.

6）得到一个"自制的"工具，它（由于其模块化的和成体系的结构）能很容易适应各种需要，例如它作为具备大量的检查出错手段的、宜于实际课堂教学使用的编译系统，或者是用于收集各种各样用户统计数据的系统.

虽然上述这几点是密切相关的，而且语言和编译程序的发展也是相互依赖的，但本文集中讨论特定语言 PASCAL 的**编译程序**的设计问题. 上述第四点特别要求这个编译程序是用它所编译的语言书写的. 因此，所要实现的编译程序不仅要求本身是高效率的，而且更主要的是它能产生高效率的代码. 这个研制项目的主要任务是要表明，这些要求是和语言里应有丰富的构造手段的愿望兼容的.

本文首先讨论 PASCAL 程序的机器代码表示方法，以及运行时程序与数据的编排，随后介绍编译程序某些部分的表示方法，特别是编译时对象的描述，以及它所用的语法分析方法. 再下一节是叙述这个编译程序研制时的**透盘考虑**，它是采用众所周知的自展技术.

自然，要在一篇短文中全面地描述这样大的课题是不可能的，而且也没必要. 它所涉及的问题和所采用的技巧，大多数都是大家知道的，而且它们的解决与改进是属于专门工程技能和训练方面的问题，并不是什么科学原则问题. 因此，本文着重讨论那些从理解整个编译程序的角度看来是新奇的或重要的方面.

1. 数据与程序的编排

在 PASCAL 中，程序结构的主要单立是过程．每个过程(或函数)都表示为一个数据段和一个程序段．**数据段**包含所有局部于该过程的可变数据，**程序段**包含不变的可执行代码和**常量**．这两种段是分开的．因此，一个过程被递归地激活时，可以使用同一个程序段；从而程序段可以由编译程序"静态地"分配．另一方面，数据段在过程每次激活时分配，在过程出口时释放．它可以采用**栈**的方法进行分配．

1.1. 数据段和变量的编址

因为分配在这个栈上的数据段可以有不同的长度，所以各数据段通过安排在它们首部的指示字(地址)链起来．这些指示字形成两条链：一条把栈上所有的数据段按照它们的生成次序链接起来；一条只把当前执行的程序段可存取的那些数据段链结起来．前者称为**动态链**（各段以它们动态生成的次序链接)，后者称为**静态链**(各段以它们在程序静态嵌套的次序链接)[4,5]．除这两个链指示字外，数据段的首部中还包含该程序在调用一个过程时所"冻结的"程序状态(地址)。数据段首部可以描述为[1]

```
type header = record
                slink, dlink: ↑stack;
                pstatus: address
              end
```

1) slink、dlink、pstatus 分别为静态链、动态链和程序状态。——译者注

编译程序把一个变量的存贮位置确定为相对于该变量所在的数据段的原点的相对地址．存取变量时，仅当特定的段的原点知道时才能计算它的绝对地址，它是从原点加上相对地址得到．

绝对地址＝段的原点地址＋相对地址

这样推迟变量位置的计算是必要的，因为(在过程递归激活的情况下）同一个数据段在栈上可以同时存在几个；而且这种推迟还有进一步好处：在任何时刻只为那些实际在使用的变量保留存贮．栈机制终将自动复盖隶属于非同时激活的过程的数据．

一个系统要真正达到高效，最为紧要的一个问题是变量地址的计算．一个系统的效率高低取决于地址计算的效率．充分地使用硬件在这方面可用的功能，显然是必要的．在有变址编址功能的计算机里，显而易见的办法是把有关系的段的原点存于变址寄存器中．可是每次存取变量还要求两次存贮访问，这个事实常常被用来作为反对递归功能的论据，认为正由于递归才一定需要这种两次访问．可是，如果一台计算机有**一组变址寄存器**可供使用，那么最好的解决办法是留出若干变址寄存器，把若干段的原点永久地(即在每个段的整个执行期间内)存放在那儿；这样，每次存取变量时只要求一次存贮访问．但是可供使用的变址寄存器通常只有为数不多的几个，因而一种实际解决办法是只把当前可引用的段（或甚至只是最近分配的那一段)的原点地址存放在变址寄存器里．这就要求每当进入一个过程（可能还有出口）时，必须将一个（或几个）变址寄存器赋以新的原点地址．适当地安排段链可以使这种"开销"维持在最低需要的水平上，少到可以忽略不计(见1.2.)．装有引用段的原点地址的寄存器的集合形成一个数组，其中第 i 个元素包含**嵌套层号**为 i 的数据段的原点，

我们把这个数组称为**区段向量** (display)[4]. PASCAL 6000 系统为此留了五个寄存器，因此允许过程的静态嵌套深度可达五层. 对于由编译程序生成的对偶〈层号，相对地址〉形式的地址，其绝对地址的计算公式是

绝对地址＝区段向量[层号]＋相对地址，

它可以用单变址指令实现.

1.2. 程 序 段

程序段由下列依次存放的小段组成：

入口代码

过程体代码

出口代码

常量

入口代码与出口代码是和数据段分配方式、"区段向量"寄存器集合的用法以及硬件寄存器的用法密切相关的. 下面是一个过程在调用与出口时所执行的整个指令序列. 它们用到以下这些变量：

P　程序状态寄存器

D　区段向量

S　栈顶指示字

T　最近生成的程序段的指针示字

W　临时变量

(在 PASCAL 6000 系统中，D、S 与 T 占用 B-地址寄存器，W占用定点工作寄存器，P 是程序地址寄存器.)

过程调用：

W := P

P := x　　转移

```
x:  T↑.pstatus := W      存程序状态
    S↑.dlink   := T      延伸动态链
    S↑.slink   := D_{n-1}  延伸静态链

    T  := S
    S  := S + s          s = 新数据段的长度
    D_n := T             确定新区段向量项目
```

第一组指令是过程调用的代码，第二组和第三组是过程入口代码．第二组指令确定新数据段首部的链值，而第三组则确定指示字S和T的新值，并确定这个新段所对应的区段向量项目．

过程出口：

```
S := T                   恢复栈顶指示字
T:=T↑. dlink             恢复段链指示字
P:=T↑. pstatus           恢复程序状态(返回转)

D_m   := T
D_{m-1}:= D_m↑.slink
   ...                   } 更改区段向量
D_n   := D_{n+1}↑.slink
```

第一组指令形成过程出口代码．第二组指令放在调用程序中紧接在过程调用代码后面，它使区段向量恢复为调用前的状态．更改运算是把静态链复制到区段向量上；所要求的更改运算数目，在$n \leqslant m$时为 $m-n+1$，否则为0．其中，

m = 调用过程的层号

n = 被调用的过程的层号

例：

```
procedure P; begin … end;
procedure Q;
    procedure R; begin … P … end;
begin … R … end;
begin {主程序}… Q … end;
```

图 1 详细地说明了这些链和指示字在 P 激活前后的状态.

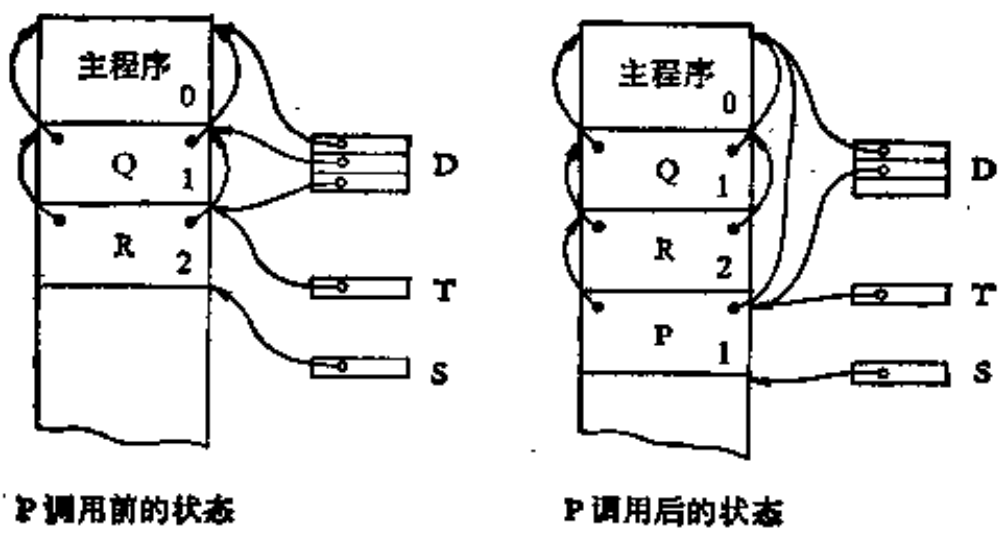


图 1　数据段的链接

1.3. 数据的表示

在设计 PASCAL 时，一个重要的决策是采用了下面的规则：一个数据段内的所有变量均可以由编译程序进行分配. 因此，排除了 ALGOL 60 那种动态数组的功能. 其好处主要是所有的地址都可以在编译时计算，并因为知道数组的维数，还可以对某些下标计算进行优化（见 1.5.）. 存贮分配问题和数据表示问题，是如何利用程序的**类型定义**来设计从抽象数据成分到计算机的存贮结构之间的映射，因为计算机的存贮结构通常是**物理存贮单元**（PSU）构成的均匀一维数组，这种单元常常称之为“字”. 如何选择这些映射多少是随实际 PSU 的大小而定的；这里所描述的是针对“字长”为 60 位的

CDC 6000 计算机的情形．一般，存取 PSU 的一部分总是比存取整个 PSU 要花费更多的时间与代码；就这点来说，这台计算机也是有代表性的．因此，整体变量总是至少分配一个全的 PSU（不紧缩）．

1.3.1. 纯量类型

纯量类型的符号常量 $C_0,\cdots,C_n$ 表示为整数 $0,\cdots,n$．

1.3.2. 数组类型

所有成分都分配整数个 PSU（数目=s）．如果 l 是下界，a 是数组变量的地址，那么它的下标为 i 的成分的地址 w 如下计算：

$$w = a + (i - l) * s = (a - l * s) + i * s = a' + i * s$$

注意：其中第一项是由编译程序计算的．多维数组可以看成是以数组结构为成分的数组．这就得到通常的下标计算多项式公式．

1.3.3. 记录类型

因为编译程序知道所有记录字段的类型，所以可以确定字段的相对位移，从而字段命名符的地址可以化为固定地址，并且不需要在运行时进行索引计算．每个字段通常占据整数个 PSU．

语法规则表明，若有变体部分，则一定跟在固定部分后面．这是保证固定部分中字段的相对位移不依赖于所选变体的必要条件．如果一个数组或记录的成分本身是个带有变体部分的记录结构，那么它的大小则是按照最大可能变体的大小选定的．如果一个记录类型定义为**紧缩的**（参考文献[1]中尚未

描述)，那么纯量、指示字或势集类型[1]的字段都分配部分的PSU. 可是，每个紧缩的字段总是分配在一个 PSU 内. 所需要的存贮位数，对定义为子域 m..n 的纯量类型是：

$$[\log(\max(\text{abs}(m), \text{abs}(n)) + 1)] + 1;$$

对定义为 $(c_0, c_1, \cdots, c_{n-1})$ 的纯量类型是：

$$[\log(n)] + 1;$$

对定义为 **powerset** m..n $(0 \leqslant m \leqslant n)$ 的势集类型是：

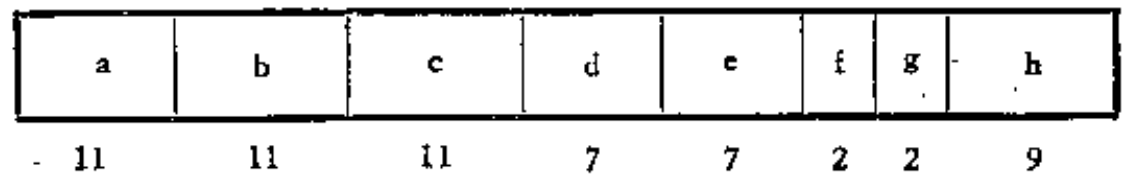
$$n + 2.$$

(为了便于使用 CDC 计算机上移位指令来截取字段，所以这里多加了一位.)

例：

```
x: packed record a, b, c: 0..999;
                 d, e: char;
                 f, g: boolean;
                 h: powerset 0..7
   end
```

这个变量 x 将以下列方式表示为一个 60 位 PSU.



我们认为重要的是，把一个记录表示为紧缩的还是非紧缩的这种选择应当留给程序员去作. 虽然存取紧缩记录的字段(特别是赋值)比较耗费时间，可是编译程序特有的对象表[2]采用紧缩结构后并未使计算时间有很大增加(多 2%)，但却使所占的存贮大大地减少(对象表少 55%，所有其他数据少 15%).

1) 在(修订)报告中已改称为"集合类型". ——译者注

2) 即我们通常所说的名字特性表. ——译者注

1.3.4. 势集类型

如果一个势集值能在一个 PSU 内得到表示，那么就可以以极高的效率实现势集运算. 这样，一个势集值含有元素 c_i 就可以用第 i 位表示. PASCAL 的集合运算是采用通常的位并行逻辑指令实现的("交"用"与"实现，"并"用"或"实现)，属于关系（即关系 in）的检验则是借助于把相应的位移到可检验的位(例如符号位)实现的.

限制势集只能由元素个数不大于计算机 PSU 长度的基集合构成，这是高效而经济地实现集合运算和势集变量的保证；其实势集变量是通常称之为位串的那种结构的独立于具体机器的抽象[6].

1.3.5. 文件类型

文件是 PASCAL 中唯一的顺序存取结构，它特别适合于分配在靠机械移动的存贮设备里. 因为辅助存贮器通常是由计算机的操作系统管理的，所以文件变量的实现就涉及到与现有操作系统的接口问题. 好在 CDC SCOPE 操作系统提供了简单高效的接口功能，于是所有 PASCAL 文件变量都分配在辅助存贮器上[8].

PASCAL 程序和 SCOPE 系统之间这种接口关系之所以简单、清晰，一个先天条件是 CDC 计算机硬件构形分为中央处理部件与外围处理部件 (CP 与 PP)；虽然这种接口关系基本上没有什么新奇之处，但还是值得提一提.

对每个文件变量，在主存里(由 PASCAL) 分配一个文件描述器和一个固定长度的缓冲区，并在辅助存贮器（盘、带或外部磁心存贮器)里(由 SCOPE)分配一个(可变长的)文件空间.

主存缓冲区与辅助存贮器之间的数据传送是由外围处理部件(PP，通道)实现的，其程序由 SCOPE 编制.缓冲区总是(至少)包含 PASCAL 文件指示字所标记的成分. 因而，元素可以在 PP 向辅助存贮器发送数据或从辅助存贮器接收数据的存贮位置上直接进行存取，并不需要象其他一般操作系统那样再耗费很大代价在内存间进行数据传送. 标准过程 **put** 与 **get** 引起的 CPU 动作只是改变指示字。这对于受输入输出限制的程序就大大地提高了效率[7].

编译程序决定缓冲区的长度 s 时基于下列两点考虑：

1) 因为整个当前文件元素 f↑ 在缓冲区内是直接可以存取的，所以 s 一定是文件成分大小 s' 的倍数. ($s = n * s'$.)

2) 因为文件在辅助存贮器里表示为有固定长度 l 的称之为物理记录单位(PRU)的序列，而且数据传输只能以整个 PRU 为单位，所以缓冲区必须至少能包含一个 PRU ($s > l$).

通常，为减少通道活动的次数，在主存很大的机器里选用由若干个 PRU 组成的缓冲区是合理的.

文件描述器内除了缓冲区的地址与长度外，还有若干指示字(见 2.2. 节). 这些指示字是缓冲区按通常的循环方式进行管理时所必不可少的[8].

每个文件成分分配整数个 PSU. 只有字符文件例外，它们在 SCOPE 系统里采用紧缩表示的方式(每个 PSU 存 10 个字符). 通常，字符文件这个概念对于书写独立于机器的程序是非常重要的，因而我们用附加的隐式例行程序实现紧缩运算与松展运算. 此时，程序员对于数据是紧缩的或非紧缩的没有明显的控制，或者甚至根本不知道，这是 PASCAL 系统中唯一的例外。

1.3.6. 类类型[1]

指示字变量以它们受囿的类变量的成分的绝对地址作为其值，这个类变量被称为是它们的**定义域**(domain)。分配给类变量一个主存区域，它的大小是由编译程序根据类类型定义里所指示的类成分的大小与成分的最大数目计算的。此外，每个类变量有一个与之相联系的地址分配指示字，它的初值是这个类区域的起始地址，每次调用标准过程

alloc (p)

时，它增加所请求的成分的大小。如果 p 的定义域的一个成分具有变体记录结构，那么其大小是由附加参数决定，这些参数用于指定新成分所希望有的具体变体。于是，类可以由大小不等的成分组成，并以最能节省存贮的方式进行分配。

指示字值 **nil** 用物理编址空间外的地址值来表示。这使得不小心以值为 **nil** 的指示字变量去引用一个成分时造成编址错误陷井，从而避免了软件去作进一步的(代价很大的)检验。没有自动的废码收集。

1) 在 PASCAL 语言的早期方案中，类 (class) 类型与指示字类型的定义如下(见[1]):

类类型指明一个由一类属于同样类型的成分所组成的结构。成分的数目是可变的。类类型变量在说明时，成分的初始数目等于 0。成分是在程序执行期间通过标准过程 **alloc** 建立(分配)的。可是，所能建立的成分的最大数目必须在类型定义里指明。

〈类类型〉::=**class** 〈最大数目〉 **of** 〈类型〉

〈最大数目〉::= 〈整数〉

指示字类型是与某个类类型变量相联系的。它的值是指向该类类型的成分的潜在指示字，指示字常量 **nil** 不指称任何成分。指示字类型被称为受囿于它的类变量。

〈指示字类型〉::=↑〈类变量〉

〈类变量〉::= 〈变量〉。——译者注

1.4. 过 程 参 数

有三种过程参数：

1）标记常量的参数[1]；

2）标记变量的参数；

3）标记过程或函数的参数.

这种体制使得除了类型不能作为参数外，为形式量所提供的对象种类和实在量可用的对象种类一样多．数据段中专门有一小段用于存放形式参数的表示.

对于常量参数，值本身如果是一个"小量"，即能存入一个 PSU 中，那么就存放在参数位置上．否则，将实在参数的地址作为形式参数的表示，存放在参数的位置上，对它的引用将是间接的．这样就避免了复制大的对象．不同于 ALGOL 60 的值参数，常量参数的定义把参数递归技术留给编译程序选择．在任何情况下，都不允许对常量参数进行赋值，编译程序可以检查出这种错误；类似于 ALGOL 60 的

```
procedure increment(x); integer x; x := x + 1;
    ···p(a + b)···
```

的情形不会出现.

变量参数将总是传递实在变量的地址，地址在该过程调用时计算.

为了唯一地表示一个过程，需要两个成分：代码的入口地址，以及局部地说明该参数过程的那个过程的数据段地址.

1) 值参数规定实参的值一律赋给形参；而这里常量参数则由编译程序根据不同情况采用直接赋值或间接引用两种办法．在 PASCAL 的后来版本中已改为值参数．——译者注

1.5. 代 码 优 化

设计这个语言，就是要使得它的极大多数成分在通常的计算机上实现时，应当有可能不需要额外的优化处理．这里介绍的编译程序把优化严格地限于上下文无关的情形．于是它不包括任何诸如查找公共子表达式（例如 a[i + j] := a[i + j] + 1）、线性下标级数[1]（矩阵计算中）、从循环中提取不变部分（例如 **for** k:=1 **to** n **do** b[i,j] := a[i,k] * a[k, j] + b[i,j]，其中 b[i,j] 的地址只需要计算一次）这类上下文有关的优化功能．最重要的一个特色是，在已知值上的计算每当有可能时都由编译程序执行；正如下面一些例子所说明的那样，这对地址计算关系极大．

对于变量

```
x:  array [10..25] of
    record  a: integer;
            b: array [-13..0] of
               record u: char; v: real
               end;
            c: boolean
    end
```

在生成的代码中，下面的变量指称都将表示为一个算好的地址：

```
x
x[13]
x[15].a
x[19].b[-7]
x[25].b[-5].v
```

1) 即递归地址计算．——译者注

另一类重要的优化是当一个运算对象为常量时的整数乘除运算. 这种优化对 CDC 6000 计算机尤为适宜，因为它一方面没有单独一条指令可以实现这种运算，而另一方面移位运算和加法运算比较快. 于是，编译程序在

x div c　　当 $c = 2^n$ $(n \geqslant 0)$ 时

的情形下利用移位指令实现；在

x * c（或 c * x）　当 $c = 2^n$，或 $c = 2^m + 2^n$，

或 $c = 2^m - 2^n$ $(m > n \geqslant 0)$

的情形下利用移位和加法运算实现. 当然，这类优化也应用于下标计算. 例如，有一变量

a: **array** [1..10, 1　20] **of** real

那么成分变量 a[i + 1, j + 1] 的地址计算只生成两条取指令（对 i 与 j）、两条移位指令和两条加法指令. 运行时，值 i*16 + i*4 + j 将作为虚变量 a[0,0] 的地址的下索引.

2. 编 译 程 序

2.1. 总 的 体 制

对一台内存相当大的高速计算机，要使编译程序高效率的关键是**一遍加工方案**. 一遍加工使存取辅助存贮器的次数减到了最少，而存取辅助存贮器要涉及到操作系统，并且耗费的时间比所有其它处理都要高一个数量级. 这个语言因编译程序采用一遍加工所强加的限制很少（对象通常必须在行文上先说明后引用）；虽则向前引用无法避免，但也没有增加很多复杂性.

PASCAL 6000 编译程序把生成的代码存放在一个主存区域里. 每个过程一编译完，就把它的代码送到辅助存贮器的

一个文件里.因为过程说明一定在语句部分前面,所以在任何时刻主存里只有属于一个过程的代码. 因此，代码缓冲区只要能放得下最长过程(通常为主程序)的整个代码就行了.

编译程序生成**绝对代码**. 在编译速度上所得到的好处，特别由于避免使用标准再定位安装程序所得到的好处，要比编译后可以合并"二进制"程序这一好处大得多，何况后者还是令人怀疑的. 如果一个编译程度的速度实际上等于再定位安装程序或联结编辑程序的速度，那么保存程序的**再定位二进制形式**就毫无意义；程序的合并应该在源程序那一级上进行. 基于这一前提的第一个软件系统(据作者所知)是 Burroughs 公司 B5500 计算机上的扩充 ALGOL[9]. 虽然该方案极其成功,却没有得到多少人拥护,其部分原因是厂商拒绝为标准语言生产较快的编译程序,另一部分原因是,如果一个新语言带有极为错综复杂的特色，那么必将使得编译程序仍然是慢的,而且还要依靠代价昂贵的优化算法.

2.2. 编译时对象的描述

程序中出现的所有标识符和其所指称的对象的描述，一起存放在一个表里. 因为刻划每个对象特征的是具有不同值域的各种各样属性，所以记录是种比较合适的数据结构. 因为各种对象是由不同的属性集合描述的，所以该记录有变体部分. 对象表本身就由这种记录的类 (class) 构成，以便采用紧缩的存贮表示,并能通过指示字快速地存取各个成分.下面给出了对象表的说明,它利用以下的数据类型[1]:

1) 为阅读方便，标识符均用汉语表示. 这里的"指示字"与"指示字型"原文均是 pointer，似乎应该区分. ——译者注

```
type 指示字=↑对象表;
     短整数 = -2^17+1..2^17-1;
     地  址 = 0..2^16-1;
     层  号 = 0..5;
     位值域 = 0..59;
     标识符类别 = (类型,常量,变量,过程,字段,标志
                   字段);
     类型形式=(数字的,符号的,指示字型,势集型,数
               组,记录,类,文件);
     标识符种类 = (实在的,形式的);
var 对象表: class of
     packed record 名字: alfa; {标识符}
     下个元素: 指示字;{链到同一作用域的下个对象}
     case 类别: 标识符类别 of
       类型: 大小: 短整数;
             case 形式: 类型形式 of
               数字的: (位大小: 位值域;
                        最小值,最大值: integer);
               符号的: (第一常量,势集· 指示字;
                        位大小 1: 位值域);
               指示字型: (定义域,元素类型:指示字);
               势集型: (基类型: 指示字);
               数组:(数组元素类型,下标类型:指示字;
                     下界,上界: 短整数);
               记录: (第一字段,记录变体: 指示字);
               类: (类元素类型: 指示字);
               文件: (文件元素类型: 指示字);
       常量: (常量类型: 指示字;
```

```
                case 常量种类：标识符种类 of
                  实在的：(后继：指示字；
                           值：integer)；
                  形式的：(常量地址：地址；
                           常量层号：层号))；
        变量：(变量类型：指示字
                变量种类：标识符种类，
                变量地址：地址；
                变量层号：层号)；
        过程：(过程类型，参数，上下文：指示字；
                过程种类：标识符种类，
                过程地址：地址；
                过程层号：层号；
                数据段大小：短整数)；
        字段：(字段类型：指示字；
                字段地址：地址；
                位移量，字段宽：位值域)；
        标志字段：(分情形大小：短整数；
                    第一变体：指示字；
                   case 标志值：  boolean of
                     false：(分情形类型：指示字)；
                     true：(分情形值：integer))
    end
```

对象表包含了全部有名对象的描述．此外，编译时还生成一些匿名对象，它们对应于成分变量的标记和因子、项、表达式等等．这些匿名对象是由局部于各个正在加工的过程的变量描述的．

匿名对象的类型称为“属性”，规定如下：

```
type 属性种类=(变元,短值,装入的值,装入的条件值);
    属性 = record 类型: 指示字;
                case 种类: 属性种类 of
                  变元: (存取: (直接,间接,变址的);
                         基: 层号;
                         位移: 短整数;
                         case 紧缩: boolean of
                           false:;
                           true:(位地址,位大小:位值域));
                  短值: (值: 短整数);
                  装入的值: (常量值: integer);
                  装入的条件值: (转: 0..3;
                                 算术: boolean)
                end
```

(如果匿名对象是一个装入的值或装入的条件值,那么用于安装它的寄存器的号实际上是知道的，因为可用的工作寄存器是作为栈处理的. 因此在属性记录中不必包含这个号，尽管它在对象的描述中是一个重要的项目.)

我们在这里完整地介绍了编译程序在描述对象时所使用的数据类型定义,这不仅为了帮助读者理解编译程序的体制，而且也想借此说明 PASCAL 的数据定义工具的能力. 它们既考虑到数据描述要采用透明的、完全符号化的、独立于机器的形式，同时也考虑到存贮器要尽可能经济地使用. 这样两个特征——系统的且适宜的抽象的可利用性，及对实际计算机高效率的适应性——的组合就是 PASCAL 设计的主要目标.

2.3. 源行文的语法分析

极大多数程序设计语言在研制时，都没有很好考虑到语言的语法结构对它们的语法分析方法的复杂性和效率的影响. 分析方法上的大量研究工作是在 ALGOL 60 发表之后进行的，因为它提出了这方面的要求. 但这些工作却增强了这样一种信念：着眼于语法限制只会妨碍语言朝着更自然、功能更强的方向发展，而且会很快变得与发展新的、巧妙的语法分析方法毫不相干.

相反，PASCAL 语法设计的主导思想是简单性，因为我们认识到，计算机难以加工的结构人们读写时也常常难以掌握；复杂的语法不仅使得分析算法比较复杂并因而效率比较低，而且也增加了人们的误解和程序编制出错的可能性. 我们还认识到，在研制一个语言的处理系统时，一个主要要求是它的有意识地诊断出语法错误，并以相当大的正确诊断概率继续加工下余行文的能力. 如果要使这个系统在程序设计的新手们中间成功地得到使用，那么对于上述这种能力必须给以最为优先的考虑. 语法分析问题从而获得了全新的方向；编译程序不仅必须加工所定义的语言，而且实际上要能加工所有由基本词汇符号组成的序列. 这个事实主要是重新确认这样的信念：所要求的并不在于为了设计一个"强功能"的语言而因此设计一个牵强附会的分析程序来设法应付该语言的所有构成成分，而在于宁可设计这样的语言，它所根据的语法可以使用相当简单而又非常清晰的分析技术.

这里选用的方法首先是由 Conway[10] 描述的，他称它为"可分转换图 (separable transition diagram)" 技术. 按照这种技术，语言的语法表示为**伪有穷状态**识别器的有穷集合. 因

为某些要识别的基本符号被替换为这个集合中某个元素可以识别的句子，所以这里加了"伪"字. 于是，这些识别器可以互相激活，可能引起递归.这种自顶向下的分析技术有下列优点：

1）每一个识别器可以用一个有穷图来表示，它清楚易懂，直捷了当地体现了识别器的程序.

2）如果程序设计系统可以使用递归过程，那么就不需要编制明显的栈机制.

3）处理语法错误用的程序路径在语法图中很容易表示，也很容易并到语法报告文件中. 虽然报告文件扩充了，但仍很简洁. 虽则一个特定的分析算法对语法的表示可能采用格式严格的表格式方法，但并不因此而使得添加出错处理部分时受到限制.

遵照这种技术，这个语言的语法描述为有穷图的集合 S，每个图有一个入口点和一个出口点. 连接两个结点的边对应于一个状态转换. 这种状态转换包括接受一个基本符号，如果那条边标以那个符号；或接受一个由 S 中某个元素可识别的句子，如果那条边标以那个元素. 语法图和 BNF-产生式之间的转换是很直捷了当的，而且验证结果语言的无二义性也不是什么深奥的问题，特别是如果 S 中没有任何元素可以接受空句子. 两个主要准则是

a） S 的每个元素是确定性的；

b）一个图的出口的选择是确定性的.

至于细节，请读者查阅参考文献[11]. 尽管看起来这些条件较为严格较受限制，但按照这些条件来塑造语言，已经证明它是相当容易的. 严格地遵守向后看一个符号这一限制所造成的与 ALGOL 60 语法的不同点，实际上只有两点是值得注意的，即

i）排除了多重赋值；

ii）排除了非整数的标号.

这样所得到的 PASCAL 语法在附录里详细说明了，它直接反映了这个编译程序的结构，差别甚微. 各个主过程间的依赖关系见图 2[1]. 和通常做法一样，不是把单个的输入字符，而是把扫描程序得到的整体符号看作基本符号. 设置扫描程序不仅可以提高效率，而且也是必要的，因为字定界符如(**begin**)的表示同标识符一样(没有换码字符),所以必须由扫描程序来“截取”. 因此，把标识符(还有数)看作基本符号. 于是值得注意的是，**面向目标的自顶向下分析程序**通过基于**自底向上**分析原则的**面向源程序的扫描**程序获得输入. Elliott ALGOL 翻译程序第一次使用了这些技术[14].

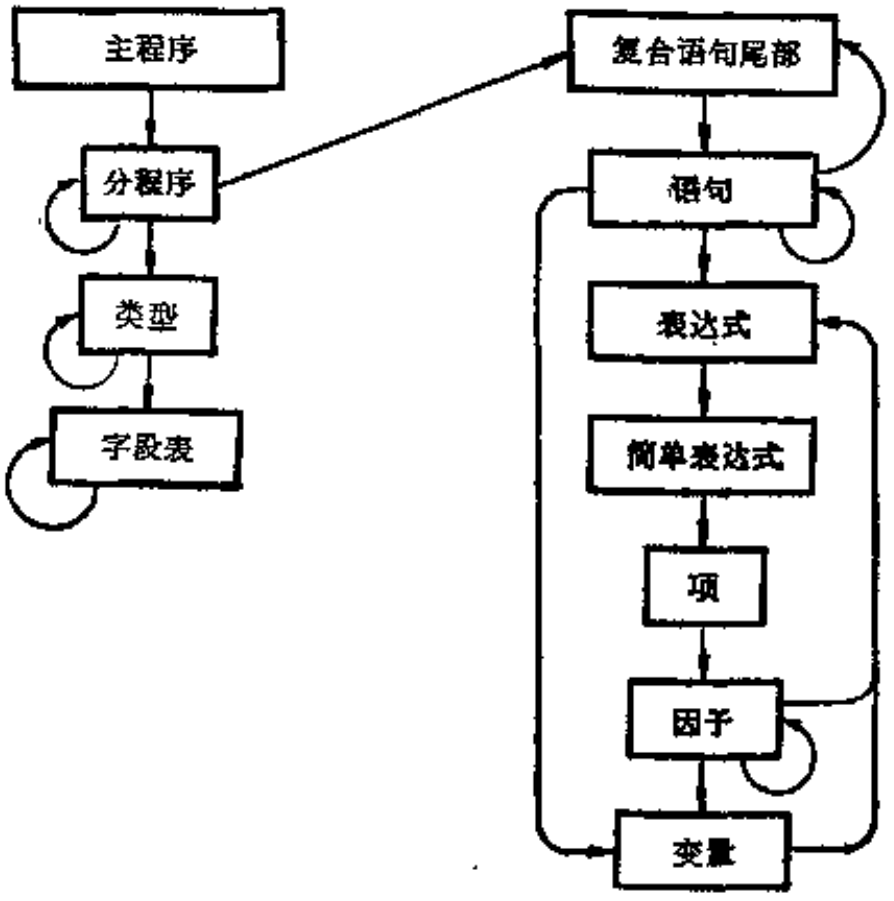


图 2　过程依赖关系图

1) 图中“复合语句尾部”是早期版本中元语言变量，参见附录. ——译者注

2.4. 性能与统计数据

2.4.1. 大小（近似数据）

源程序：

总共	4000	行
其中包括：	130,000	字符(不包括注释)
	41%	空白符
	41%	字母
	3%	数字
	15%	特殊字符
源程序每行平均长度：	33	字符

该程序共有500个不同的标识符，平均每个标识符出现18次。字定界符出现5000次，频率最高的是 **end**、**begin**、**if**、**then** 与 **else**，它们分别出现850、810、780、780、400次。

目标程序：

需要的区域总长度[1]	19,000 (=45,000$_8$)	字
编译程序本身：	67.8%	
目标代码缓冲区[1]：	4.7%	
对象表[1]：	9.2%	
其他数据：	4.5%	
输入输出缓冲区：	8.3%	
接口和I/O例行程序：	5.5%	

程序总共有32,700条指令，它们分布如下：

1)	48.7%	长指令(30位)
	28.7%	短指令(15位)

1) 已调正过，使得可以编译这个编译程序本身。——原注

	22.8%	充填指令（空操作）
	100%	
2）	27.6%	存取指令
	15.0%	装入直接量的指令
	3.5%	算术指令
	14.4%	逻辑和移位指令
	6.2%	基地址寄存器指令
	10.5%	转移和子例行程序调用
	22 8%	充填指令（空操作）
	100%	

3）X-寄存器在表达式求值时是保存中间结果用的栈。编译结果有7,927条指令是给X-寄存器赋值的，它们分布如下：

X1	75.0%
X2	21.6%
X3	3.1%
X4	0.3%
X5	0.02%

4）B-寄存器用作区段向量D（参见1.2.）。编译结果有1,564条指令是用于修正区段向量的，它们的分布如下：

B1	84.2%
B2	14.7%
B3	1.1%
B4	未用

CDC 6000计算机提供的转移控制只能转到字的开始；这就说明，对于转移指令百分比很高的程序，具有大量的充填指令是它的一大特征。

2.4.2. 速度

在 CDC 6000 计算机上重新编译这个编译程序时测量了它的性能，数据如下(如果编译较短程序结果会更好一些)：

安装和编译(该编译程序的源程序)的时间

40 秒 (CP) + 15 秒 (PP)

由此得到平均值

100　　每 (CP) 秒加工的源代码行数

820　　每秒生成的指令条数.

(对 CDC 6600 的期望改进因子大约是 2.5.)

3. 编译程序设计技术

PASCAL 早期文本的实现工作是在 1968 年开始的. 编译程序用 FORTRAN 书写，打算以后翻译为 PASCAL 语言，用自展方法实现. 结果发现这是个大错误. 虽然编译程序是完成了，但还是决定从头做起研制一个新的编译程序，并利用这个机会对语言定义的某些细节作些修改. FORTRAN 这个语言显然不宜用来描写复杂的程序结构和数据结构，用这样一种语言书写编译程序的结果，必定是为了符合 FORTRAN 的规则而使得数据的定义 (并因此还有程序) 非常牵强附会，以致于书写一个新的编译程序比"翻译"这个已有的程序要容易得多.

第二个编译程序是在 1969 年末开始研制的，它**一开始就用 PASCAL 书写**. 这意味着该语言一定已经过严格的定义，因为即使对语言作极小的变动也必将使得修改编译程序的工作量相当大. 编译程序用 PASCAL 书写之后，用"手工"把它翻译为可用的低级语言，并未企求做任何优化. 这个翻

译工作以及调试这个编译程序使其达到能编译它自己原来版本的程度，大约需要一个人-月的工作量．这个阶段省略了对于表这编译程序看来是不那么重要的若干特色：势集结构、紧缩记录、函数、过程参数和函数参数、实数算术运算以及所有的局部代码优化．尽管如此，这个初始版本的编译程序和由它经过多次自展阶段所得的最终版本相比，肯定被认为是相当大的(>60%)．这个程序将近有3,000行，书写时没有机会检验——这是任何熟练的软件工作者一种正常的工作情况．从一个很小的初始文本着手工作的好处是，最初把它手工翻译为可用的语言时所花费的工作量最小．可是，这种好处很少，而且在以后的自展阶段中为了把这一些特色付诸实用将要偿还更多的工作量．经验证明，为了使新特色付诸实用而重写编译程序的工作常常由于其它更为紧迫的事务而搁置下来，工作量越积越大，直到最后觉得代价太大了不值得去做．因此，如果某种特色本身相当重要，而且搁到以后再引进则需要相当大的工作量来重写，那么一开始就用包含所有这种特色的语言进行工作总是合算的．值得注意的是，特别是记录结构、变体记录结构、文件结构和类结构都应当包括在初始的编译程序版本中．

这个编译程序是由三人小组研制的，他们的任务划分如下：

1）类型定义、变量说明以及包括形式参数表在内的过程首部；

2）表这式和语句；

3）与操作系统的接口．

这样划分任务的结果使得各部分之间的相互关系最少．第一、二两部分之间的主要接口是对象表（参见2.2.），它由第一部分的程序构造，并由第二部分的程序检索使用．因此，确定对

象表结构必定是最早并且最有影响的设计决定之一．在这样的软件工程课题中，适当地划分任务是协同设计和管理方面的一种较为紧要的决策，并且对所要进行的整个工作的数量和质量具有决定性的影响．按照上述那种划分，这个编译程序大约需要两个人-年的工作量．

选择这种方法研制编译程序——用语言本身而不是用一个通常可用的语言书写初始的版本——对选用何种技术把编译程序移植到其他的计算机上有一定的影响.先用FORTRAN书写从而使它自动在几乎所有的计算机上可以使用——这一思想是产生第一个（失败了的）设想的动机．虽然初看起来它是很有吸引力的，但它由于下列事实而受到妨碍．实际上，在不同的计算机上的 FORTRAN 文本远不是兼容的，以致于移植一个大的程序已经是个很大的负担——且不说这种编译程序非常含糊，要使得能够产生不同的目标代码而只作很小的变更，是极其困难的．

移植一个 PASCAL 编译程序（在除 PASCAL 本身外没有任何语言可用的情况下)的一个显而易见的方法，是首先重写代码生成程序和它们的调用，使其生成目标计算机的代码(不考虑优化)．第一步完成之后，在第二步中可以用这个新的编译程序来自展本身，于是在宿主计算机A上生成了目标计算机B的二进制版本，它很容易再被移植（见图 3)．目前正在用这个方法把编译程序移植到 ICL 1900 计算机上．

如果在计算机 B 上工作的实现者没有计算机 A 可供使用，那么上述第二步可以改为：把第一步的结果手工翻译为B计算机上一种可用的语言L的等价程序．如果L提供了递归过程，那么翻译工作将大大简化．所用的语言L的翻译程序的实际效率并不影响从自展过程得到的最终版本．目前正在用这种方法，利用 PL/1 把编译程序移植到 IBM 系统/360

计算机上.

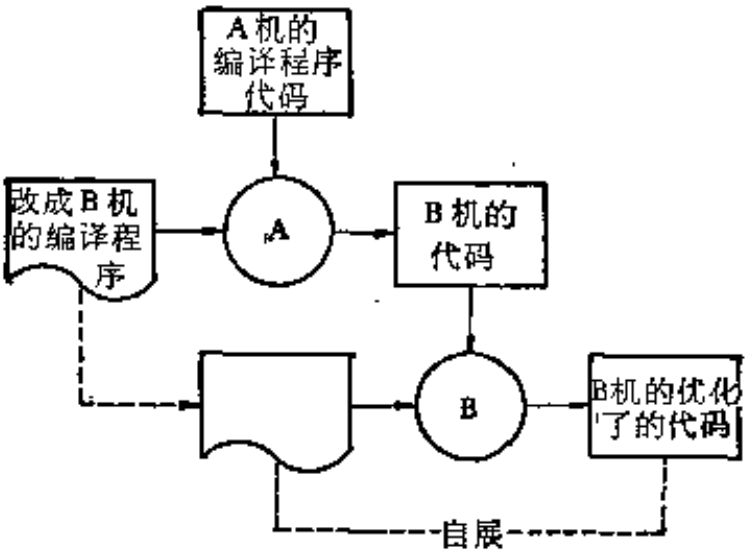


图 3　自 展 步 骤

4. 编译复杂性和计算机总体结构的关系

确实，在设计 PASCAL 语言时，我们没有考虑到任何具体的机器. 目的是要得到一种语言，它的程序对于所有的常规机器都能不花费很大代价就可翻译为原来程序的高效代码. 这里值得研究的是：假若要使代码编译得好又要使所费的代价少，那么计算机的总体结构应该具有什么样的特性才最为合适，特别是以这些准则衡量一下 CDC 6000.

能生成高效目标代码的一个极为重要的尺度是至少存在两个变址(或基址)寄存器. 另一个更为一般的特征是指令系统和整个总体结构都要非常简单. 设备愈多，例如可以使用几套寄存器，那么为了用好这些设备，编译处理也就愈复杂. 真正好的总体结构将会使得所谓的优化基本上没有必要（或不可能）. 程序分析方面的最新结果表明，在极大多数的程序里最大量的时间花费在极小百分比的代码中，于是在编译时作复杂的全局优化是极其浪费的[12]. 而且，甚至有选择的优

化也要求编译程序必须有完全的优化例行程序，所以一种总体结构如果能够把这种优化省去，那是最吸引人了.

CDC 6000 总体结构的主要优点是它指令系统整齐、简洁. 出乎一般人的预料，扩充指令系统并不增强一台计算机的能力（至少对编译程序的用户和设计者不是这样），它反而表明设计师缺乏一致的、清楚的概念. 功能的多种多样性由软件来提供要比硬件更为经济一些，当前热门的“微程序设计”以及把高级语言直接编译为“微代码”[13] 的经验已证实了这一点. 虽然 CDC 6000 计算机只有为数不多的 64 条指令，但编译程序在生成代码时只用了 42 条指令（占 66%）；对于指令系统精致的计算机，可以指望百分比还要小得多. 图 4 表示了不同指令的数目和这些指令在编译程序的代码里所占百分比之间的关系. 它表明，只要四条指令就构成了整个代码的 64%（包括用得最多的空操作指令）.

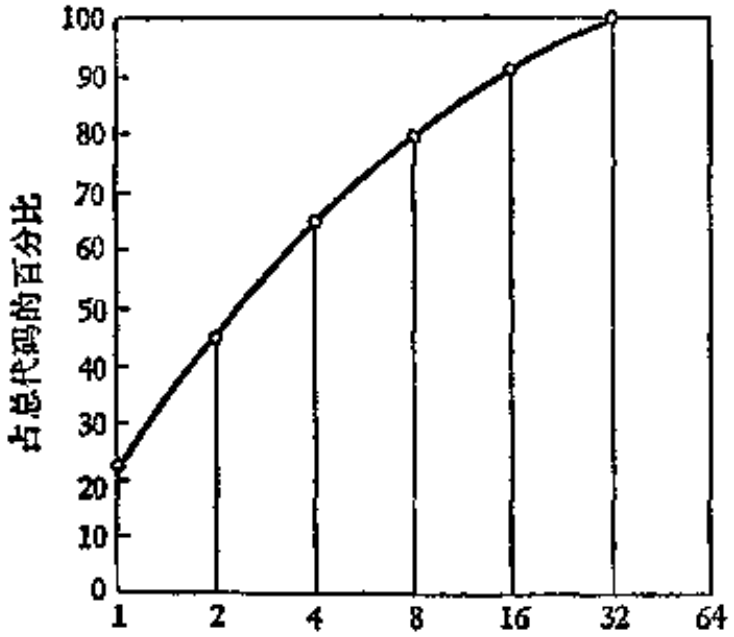


图 4　指令使用情况

考虑到 CDC 6000 计算机已经很好地满足了上述两个主要要求，它的缺陷看来还是比较小. 但是有些东西在一定的

情况下使用起来仍然很不顺手．关于这方面，我们特别指出以下几点：

1）缺少一种能把当前处理机状态存入通用寄存器的转移指令（RJ 指令不能用于可重入代码）；

2）缺少逻辑移位指令；

3）因为采用反码算术运算，0 有两种表示．

CDC 6000 计算机的一个大缺陷是对于整数溢出情况缺少自动陷井，它全然不管这种情况，也没有任何办法可以检测到．这确实比较恼火，因为有些指令在 60 位整数上运算，有些在 48 位整数上运算，有些在 18 位整数上运算，有些从一种表示扩展为或截取为另一种表示，虽然它们正确地处理了符号位，但是没有检查溢出．这个极其严重的缺陷使得整数计算相当危险．必须注意到，某些其他计算机也存在这种"特征"，而且其影响也不只是对 PASCAL．希望未来计算机的——或扩充现有计算机的——设计师将对于这样一些看来是微不足道的特性能给予较多的注意．

5. 结　论

这个编译程序的经验证实了下列结论是正确的，即设计一个有足够内容的而语法上简单的语言和研制一个没有附加优化特点的编译程序——这样两个原则是合理的．可是各种各样用户小组会根据各自的愿望——特别是请求——提出各种扩充；如果屈从于他们的压力，那么这种成功也很容易转化．在多道程序系统里，编译程序是否可以在高优先级别上使用的关键是它的大小要"合情合理"．如果已经比四分之一的内存多了一点，那么我们建议要小心谨慎，不要再做任何进一步的扩充．

根据经验来看，另一个设计决定也证明是正确的，即根据可以使用的寄存器数目来限制表达式求值的栈与区段向量．于是，如果一个程序在计算某个表达式时需要五个以上的中间结果或者如果过程嵌套层数大于五层时，编译程序应当产生一种信息，而不必提供一种“软件解决”，因为这种软件解决要求编译程序增加大量的（很少使用的）例行程序，并意味着它的代码效率很低．属于同一范畴的另一项设计决定是排除用复杂的例行程序来保证结果表达式所使用的寄存器最少；尽管这种例行程序极少有机会做有意义的优化，但却会加剧编译程序的扩展，并进一步增加复杂性[12]．看来，设计一个好的编译程序的关键是要有一种辨别能力，或者直观地知道什么是重要的，什么是微不足道的．这种工程上的直觉观念比起诸如语法分析理论这类渊博的学问来，更为至关紧要．

下面四个程序是用 ALGOL、FORTRAN、与 PASCAL 三种语言书写的，并都在 CDC 6400 计算机上执行了，以便粗略地比较一下所生成的代码的质量．选择这些程序是想使得各语言之间可以明显而直捷了当地进行翻译．因此，这些程序并不能说明 PASCAL 的表达能力，它们也被限于这三种语言多少都可方便表达的范围里．

1）矩阵乘 B:=A*A，无输出．

2）一个有 2000 个数的数组的分类．

3）寻找整数 1 至 30 的所有可能的加法分析(利用递归，在 FORTRAN 中用“手编”的栈)．

4）文件中字符的计数．

对所有程序，为编译(可忽略)和执行所需要的中央处理机时间总和如下[1]：

1）表中“sec”为秒，“rel.”为相对于 FORTRAN IV 的时间比．——译者注

程　　序	ALGOL 60		FORTRAN IV		PASCAL	
	sec.	rel.	sec.	rel.	sec.	rel.
矩阵乘 n = 100	115.1	5.82	19.8	1	48.5	2.45
					32.1[1]	1.62
矩阵乘n=64=2^6	32.2	5.85	5.5	1	8.6	1.56
分类 n = 2000	150.0	4.38	34.3	1	40.0	1.17
分拆 n = 30	106.8	8.75	12.2	1	9.28	0.762
分拆,无输出	10.0	6.00	1.66	1	1.20	0.722
数字符个数	302.2[2]	8.24	36.7	1	7.32	0.20
	360.7[2]	9.83				

1) n = 96 = $2^6 + 2^5$ 的情形.

2) 302.2 是利用 IFIP 标准I/O过程 **incharacter**; 360.7 是利用"ACM-Knuth-I/O" 过程 **output.**

```
'BEGIN' 'COMMENT' MATRIX MULTIPLICATION;
    'INTEGER' I, J, K, N; 'REAL' S;
    'REAL' 'ARRAY' A, B[1:100, 1:100];
    N := 100;
    'FOR' I := 1 'STEP' 1 'UNTIL' N 'DO'
    'FOR' J := 1 'STEP' 1 'UNTIL' N 'DO' A[I, J]:=1.0;
    'FOR' I := 1 'STEP' 1 'UNTIL' N 'DO'
    'FOR' J := 1 'STEP' 1 'UNTIL' N 'DO'
    'BEGIN' S := 0;
        'FOR' K:=1 'STEP' 1 'UNTIL' N 'DO'
              S := A[I, K]*A[K, J] + S;
        B[I, J] := S
    'END'
'END';
```

```
      FORTRAN IV PROGRAM MATMULT (OUTPUT)
      DIMENSION A(100, 100), B(100, 100)
      N = 100
      DO 10 I = 1, N
      DO 10 J = 1, N
   10 A (I, J) = 1.0
```

```
      DO 30 I = 1, N
      DO 30 J = 1, N
      S = 0
      DO 20 K = 1, N
   20 S = A(I, K) * A(K, J) + S
   30 B(I, J) = S
      STOP
      END
```

```
! MATRIX MULTIPLICATION''
    CONST N = 100;
    VAR I, J, K; INTEGER; S: REAL;
        A, B: ARRAY [1..N, 1..N] OF REAL;
BEGIN FOR I := 1 TO N DO
      FOR J := 1 TO N DO A[I, J]: = 1.0;
      FOR I := 1 TO N DO
      FOR J := 1 TO N DO
      BEGIN S := 0;
          FOR K := 1 TO N DO S := A[I, K]*A[K, J] + S;
          B[I, J] := S
      END
END.
```

```
'BEGIN' 'COMMENT' SORTING AN ARRAY;
    'INTEGER' I, J, K, M, N;
    'INTEGER' 'ARRAY' A [1:2000];
    N := 2000;
    'FOR' I := 1 'STEP' 1 'UNTIL' N 'DO'
        A[I] := I;
    'FOR' I := 1 'STEP' 1 'UNTIL' N - 1 'DO'
    'BEGIN' K := I; M := A[I];
        'FOR' J := I + 1 'STEP' 1 'UNTIL' N 'DO'
        'IF' A[J] > M 'THEN' 'BEGIN' K := J; M := A[J] 'END';
        A[K] := A[J]; A[I] := M;
    'END'
'END';
```

```
      FORTRAN IV PROGRAM SORT (OUTPUT)
      INTEGER A(2000)
      N = 2000
      DO 10 I = 1, N
   10 A(I) = I
      N1 = N - 1
      DO 30 I = 1, N1
      K = I
      M = A(I)
      I1 = I + 1
      DO 20 J = I1, N
      IF (A(J) .LE. M) GO TO 20
      K = J
      M = A(J)
   20 CONTINUE
      A(K) = A(I)
   30 A(I) = M
      STOP
      END
```

```
! SORT AN ARRAY OF INTEGERS"
    CONST N = 2000;
    VAR I, J, K, M: INTEGER;
        A: ARRAY [1..N] OF INTEGER;
    BEGIN FOR I:=1 TO N DO A[I]:=I;
          FOR I:=1 TO N - 1 DO
          BEGIN K:=I; M := A[I];
              FOR J:=I + 1 TO N DO
              IF A[J] > M THEN BEGIN K := J; M := A[J] END;
              A[K] := A[I]; A[I] := M
        END
END.
```

```
'BEGIN' 'INTEGER' 'ARRAY' P[1:100];
    'PROCEDURE' A(M, N, I);
        'VALUE' M, N, I; 'INTEGER' M, N, I;
```

```
'BEGIN' 'INTEGER' Z, K;
    'FOR' Z:=1 'STEP' 1 'UNTIL' M 'DO'
    'BEGIN' P[I] := Z;
        'IF' Z = N 'THEN'
        'BEGIN' OUTPUT(61, '('/, 40(BZD)')', (P[K], K :=1:I));
        'GOTO' EX;
        'END';
    A (Z, N - Z, I + 1);
    'END' FOR;
EX: 'END' A;

    A (30, 30, 1);
'END';
```

```
      PROGRAM PART (OUTPUT)
      INTEGER ZZ(100), NN(100), MM(100), II(100), P(100)
      INTEGER M, N, I, Z, S
      M = 30
      N = 30
       I = 1
       S = 1
    1 Z = I
    2 P(I) = Z
      IF (Z - N) 10, 20, 10
   20 PRINT 60, (P(K), K = 1, I)
   60 FORMAT(1X, 40 I 3)
      GO TO 30
   10 ZZ(S) = Z
      NN(S) = N
      MM(S) = M
      II(S) = I
      S = S + 1
      M = Z
      N = N - Z
      I = I + 1
      GO TO 1
```

```
   30  S = S - 1
       IF (S) 31, 31, 32
   32  Z = ZZ(S) + 1
       N = NN(S)
       M = MM(S)
       I = II(S)
       IF(Z - M) 2, 2, 30
   31  STOP
       END
```

```
   VAR P: ARRAY [1..100] OF INTEGER;
   PROCEDURE A(M, N, I: INTEGER);
      VAR Z, K: INTEGER;
   BEGIN FOR Z := 1 TO M DO
         BEGIN P[I] := Z;
            IF Z = N THEN
            BEGIN FOR K:=1 TO I DO WRITE(P[K]);
                  WRITE(EOL); GOTO 1
            END;
            A(Z, N - Z, I + 1)
         END;
   1: END;
BEGIN A(30, 30, 1); WRITE(EOL)
END.
```

```
'BEGIN' 'COMMENT' COUNT CHARACTERS IN FILE;
   'INTEGER' I, J, Z42;
   'INTEGER' 'ARRAY' C[-1:63];
   'FOR' I := -1 'STEP' 1 'UNTIL' 63 'DO' C[I]:=0;
   EOF(60, L2); Z42 :=2^42;
L1: INCHARACTER (60,
'('ABCDEFGHIJKLMNOPQRSTUVWXYZ 0123456789
          +-*/( )$=, .'[]:#!%@^''< >@\_' ')', I);
   O[I] := C[I] + 1;
   'GOTO' L1;
L2: 'FOR' I := 0 'STEP' 1 'UNTIL' 63 'DO'
OUTPUT(61, '('A,9ZD/')', I * Z42, C[I]);
```

```
'END';

'BEGIN' 'COMMENT' COUNT CHARACTERS IN FILE;
    'INTEGER' I, J, Z42;
    'INTEGER' 'ARRAY' C [-1:63], A [1:80];
    'FOR' I:=-1 'STEP' 1 'UNTIL' 63 'DO' C[I] := 0;
    EOF(60, L2); Z42 := 2^42;
L1: INPUT (60, '('80(A)')', A);
        'FOR' J := 1 'STEP' 1 'UNTIL' 80 'DO'
        'BEGIN' I := A [J]//Z42; C [I] := C [I] + 1;
        'END';
    'GOTO' L1;
L2: 'FOR' I:=0 'STEP' 1 'UNTIL' 63 'DO'
        OUTPUT(61, '('A,9ZD/')', I * Z 42, C[I]);
'END';
```

```
      FORTRAN IV PROGRAM COUNT (INPUT, OUTPUT, TAPE1
     =INPUT)
      INTEGER A(80), C(64)
      DO 10 I = 1, 64
   10 C(I) = 0
   20 READ 100, (A(J), J = 1, 80)
      IF (EOF, 1) 50, 30
   30 DO 40 J = 1, 80
      I = A(J) + 1
   40 C (I) = C (I) + 1
      GO TO 20
   50 DO 60 I = 2, 64
      K = I - 1
   60 PRINT 201, K, C (I)
  100 FORMAT(80R1)
  201 FORMAT(1H, R1, I10)
      STOP
      END
```

" COUNT CHARACTERS IN FILE"

```
    VAR I: CHAR;
        C: ARRAY [CHAR] OF INTEGER;
BEGIN FOR I := EOL TO ';' DO C[I] := 0;
      READ(I);
      REPEAT C[I] := C[I] + 1; READ (I)
      UNTIL EOF(INPUT);
      WRITE('EOL', C[EOL], EOL);
      FOR I := 'A' TO ';' DO WRITE(' ', I, C[I], EOL)
END.
```

也许比这个系统的效率更为重要的是它可靠性的程度。该系统的可靠性主要是通过两个途径取得的：即使用简单的、清晰的语法分析方案；用 PASCAL 语言本身来描述编译算法和基本的数据结构。从而，语言的各个特色的编译可以分别检验，并在去掉大部分笔误之后，保证各有关例行程序的正确性就有很大的把握。之所以可以选用"递归子程序"方法进行语法分析，同样也是由于我们可以使用一个象 PASCAL 那样的有递归过程的语言。经验证明，选择递归子程序是好的。用这个方法进行扩充、更改出错诊断功能，已证明是比语法表的方法——如该编译程序的第一版本所用的优先分析方案——容易。这种技术的一个有趣的，可能也是很有意义的副作用是，语法本身用框图描述，不是用 BNF 产生式描述。这些图直接表示了编译程序的控制结构。每个图对应一个过程。因为这种图是正规结构，所以它们都可以用如果语句、分情形语句、当语句和直到语句来表示。在这个编译程序里，转向语句只用于处理源程序的语法错误。这是提高编译程序的可读性和可靠性方面的最为重要的一个事实，当然这也要求对部分程序员进行大量的训练。但是，几乎同样重要的是PASCAL的丰富数据构造工具的可用性(见 2.2.)，它使 PASCAL 成为一个面向问题的并独立于机器的语言，同时使数据有高效而经济的表示方法。于是，许多方面都证明了 PASCAL 是个系

统地描述大而复杂的系统程序的必要而理想的工具。

谢　　词

作者谨向合作者 U. Ammann、E. Marmier 和 R. Schild 表示感谢，他们的鼓励和辛勤工作是整个 PASCAL 项目研制成功的必不可少的条件。

参 考 文 献

[1],[2]同本书第二部分参考文献的[1]与[2]。

[3] E. W. Dijkstra, Notes on Structured Programming, EWD 249, Technical University Eindhoven, 1970.

[4] E. W. Dijkstra, 'Ein ALGOL-60 Uebersetzer für die X1', MTW, 8, 54～56, 115～119, 1961.

[5] B. Randell and L. J. Russell, ALGOL 60 Implementation, Academic Press, London, New York, 1964.

[6] C. A. R. Hoare, Set Manipulation, ALGOL Bulletin, 27.3.4, 1968.

[7] C. A. R. Hoare, File Processing, ALGOL Bulletin, 25.3.3, 1967.

[8] Control Data Corporation, SCOPE Reference Manual, Publication No. 60189400.

[9] Burroughs Corporation, Extended ALGOL Reference Manual, Publication No. AA 37688.

[10] M. E. Conway, 'Dasign of a separable-transition-diagram compiler', *Comm. ACM*, **6**, 7, 396～408, 1963.

[11] D. E. Knuth, Top-down Syntax Analysis, International Summer School on Computer Programming, Copenhagen, 1967.

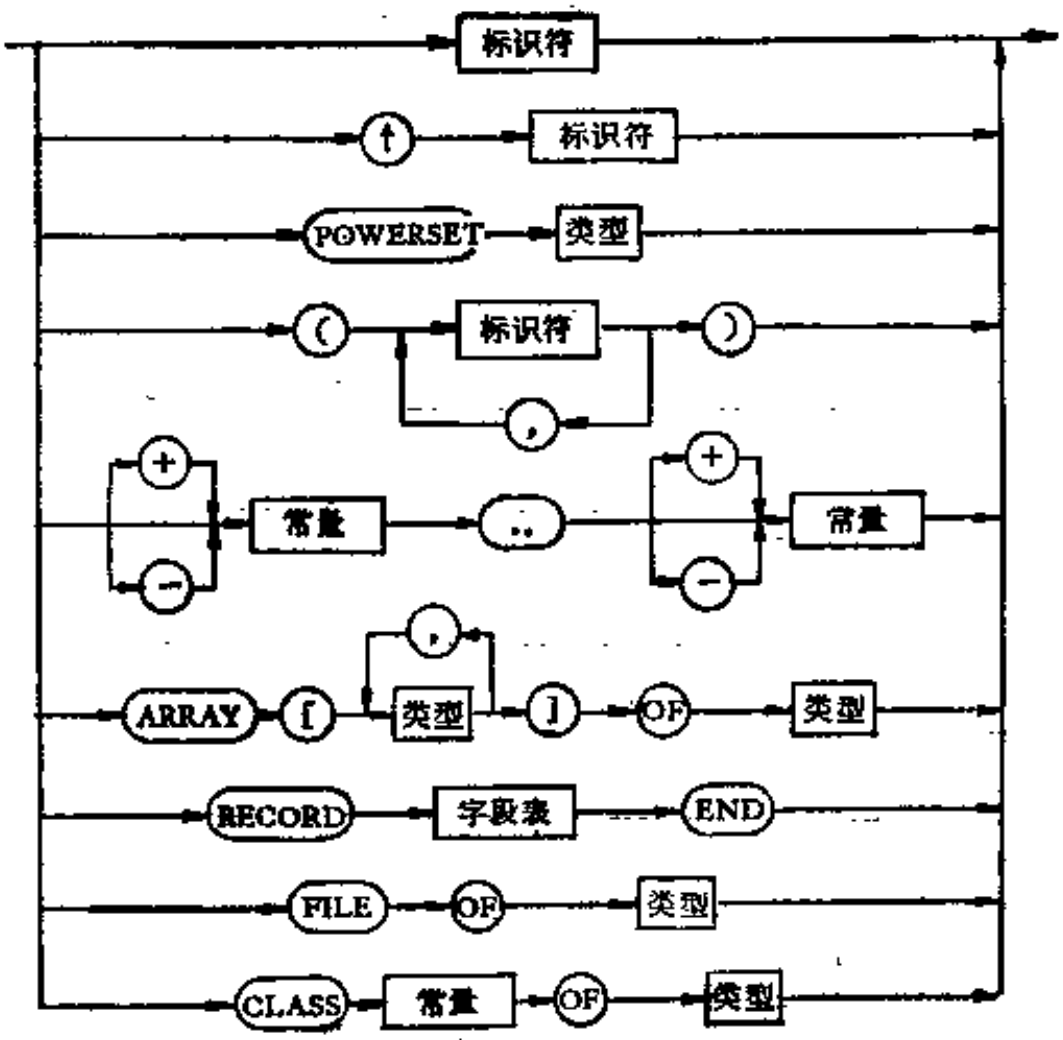
[12] D. E. Knuth, 'An empirical study of FORTRAN programs', *Software——Practice and Experience*, 1, 105～133, 1971.

[13] H. Weber, 'A microprogrammed implementation of EULER on IBM system/360 model 30'. *Comm. ACM*, **10**, 9, 549～558, 1967.

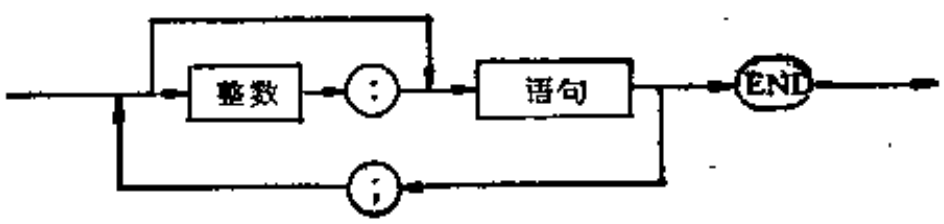
[14] C. A. R. Hoare, 'The ELLIOTT ALGOL programming system', in Systems Programming, Ed. P. Wegner, Academic Press, London, New York, 1964.

附录　语法图

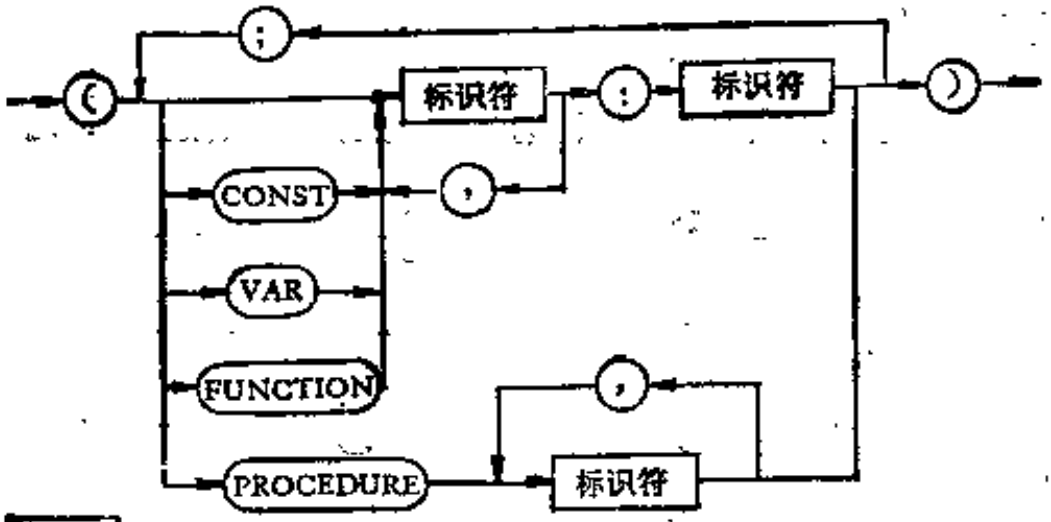
类型



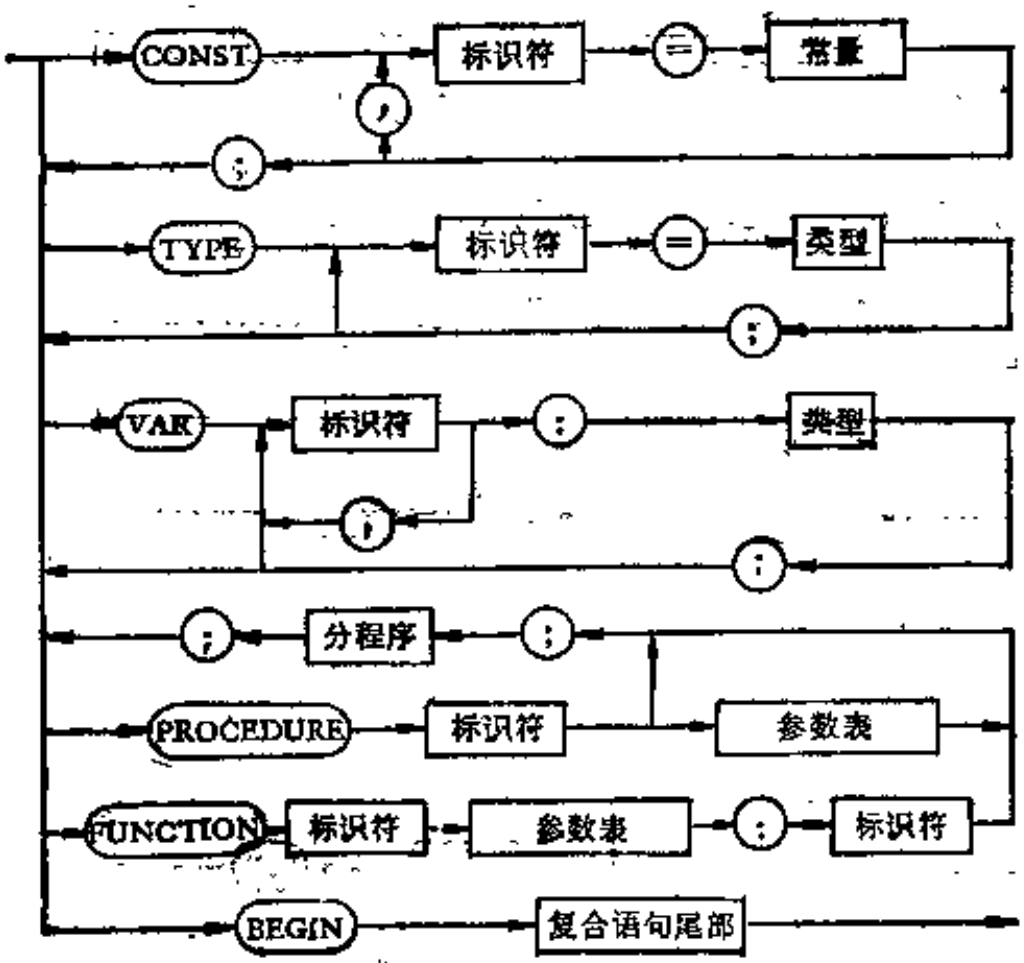
复合语句尾部



参数表


(
;
标识符
,
:
标识符
)
CONST
VAR
FUNCTION
PROCEDURE
,
标识符

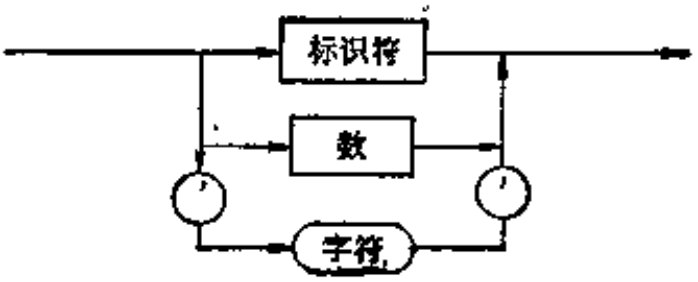
分程序


CONST
标识符
=
常量
,
;
TYPE
标识符
=
类型
;
VAR
标识符
:
类型
,
;
;
;
分程序
;
PROCEDURE
标识符
参数表
FUNCTION
标识符
参数表
:
标识符
BEGIN
复合语句尾部

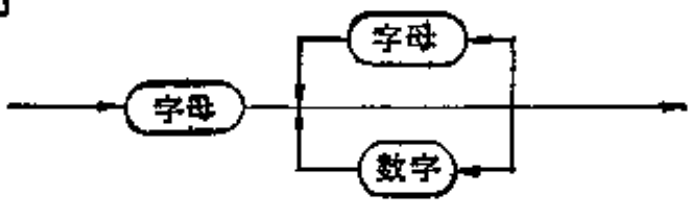
程序



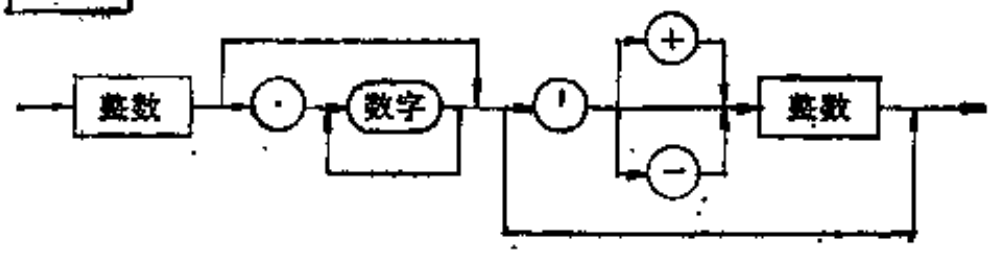
常量



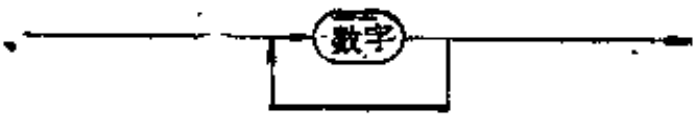
标识符

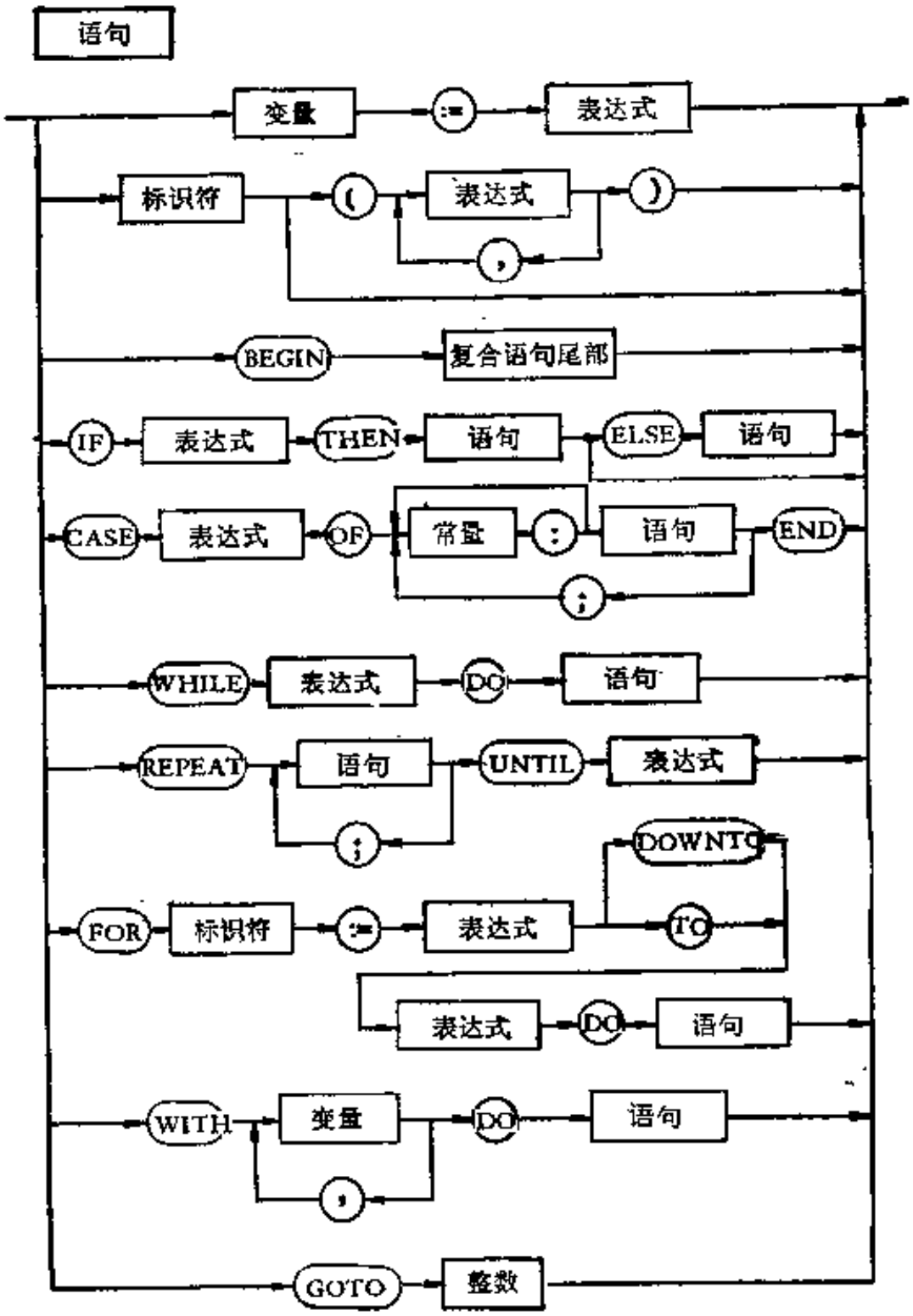


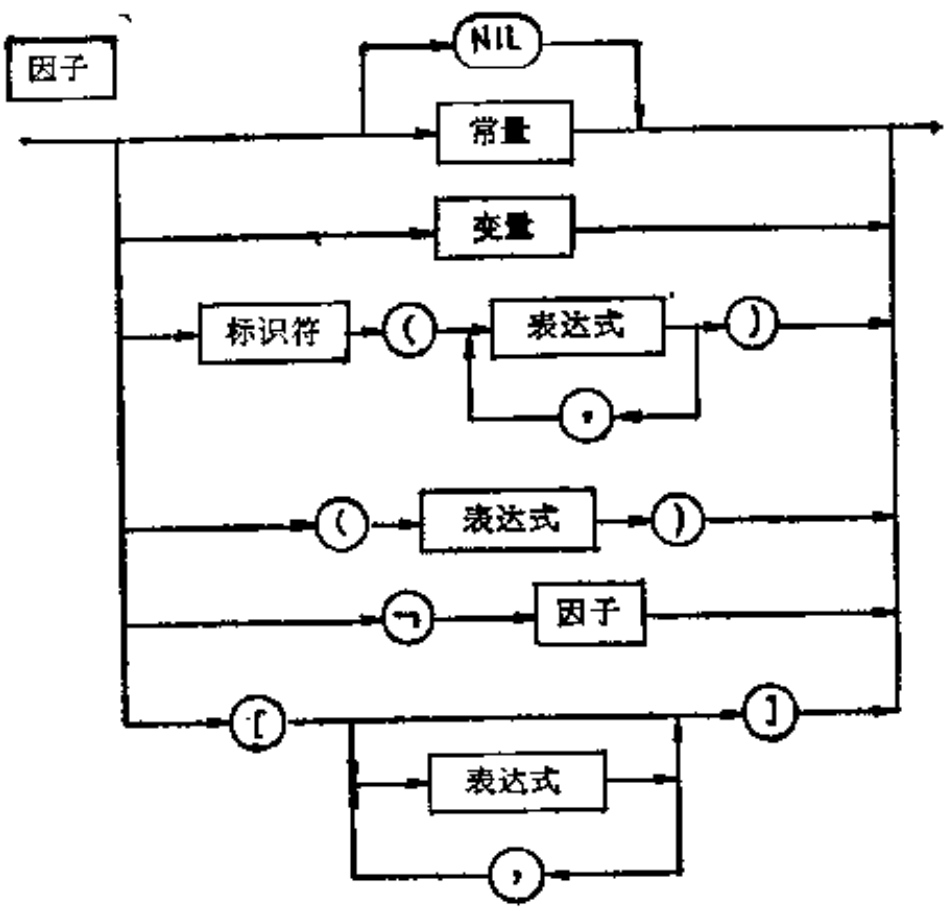
数

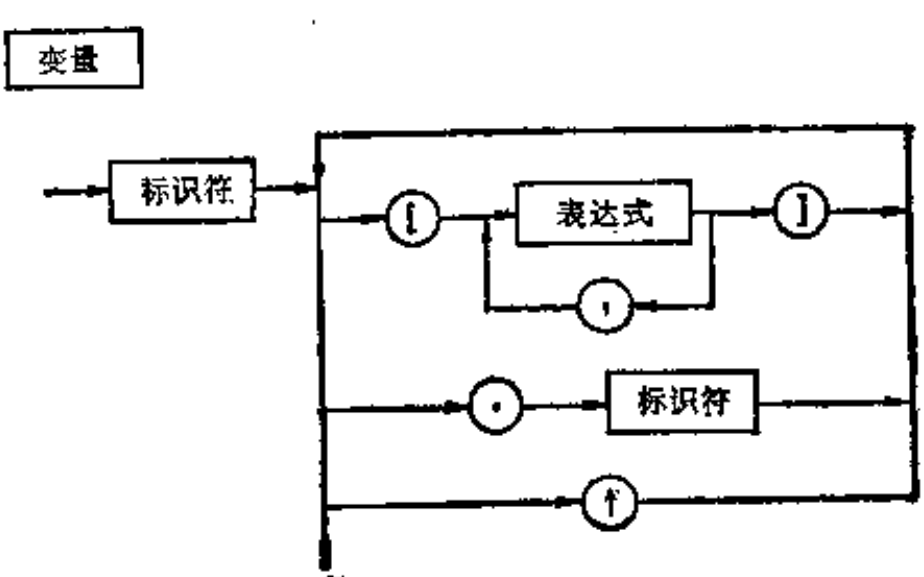


整数

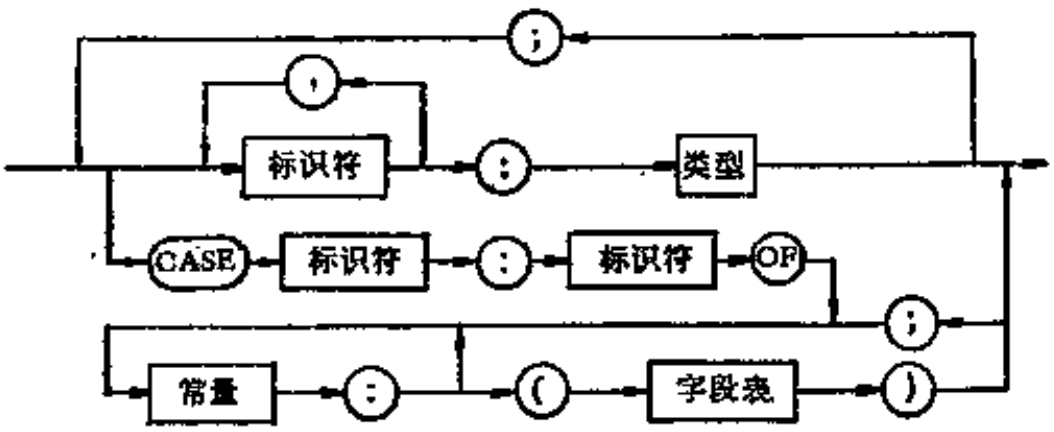



语句
变量
:=
表达式
标识符
(
表达式
)
,
BEGIN
复合语句尾部
IF
表达式
THEN
语句
ELSE
语句
CASE
表达式
OF
常量
:
语句
END
;
WHILE
表达式
DO
语句
REPEAT
语句
UNTIL
表达式
;
DOWNTO
FOR
标识符
:=
表达式
TO
表达式
DO
语句
WITH
变量
DO
语句
,
GOTO
整数

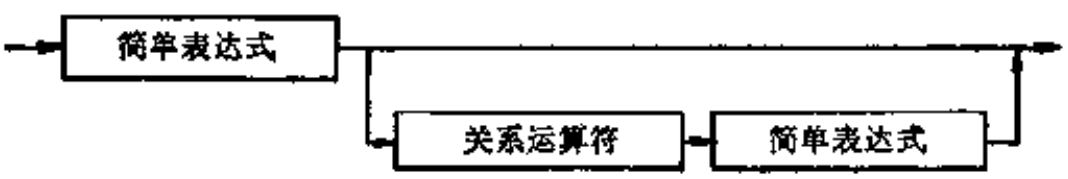

因子
NIL
常量
变量
标识符
(
表达式
)
,
(
表达式
)
¬
因子
[
]
表达式
,


变量
标识符
[
表达式
]
,
.
标识符
↑

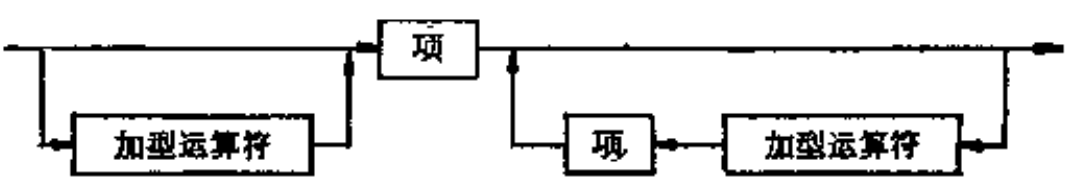
字段表



表达式



简单表达式



项

